Chitta Baral
Gianluigi Greco
Nicola Leone
Giorgio Terracina (Eds.)

LNAI 3662

Logic Programming and Nonmonotonic Reasoning

8th International Conference, LPNMR 2005
Diamante, Italy, September 2005
Proceedings

Lecture Notes in Artificial Intelligence 3662

Edited by J. G. Carbonell and J. Siekmann

Subseries of Lecture Notes in Computer Science

Chitta Baral Gianluigi Greco
Nicola Leone Giorgio Terracina (Eds.)

Logic Programming and Nonmonotonic Reasoning

8th International Conference, LPNMR 2005
Diamante, Italy, September 5-8, 2005
Proceedings

Springer

Series Editors

Jaime G. Carbonell, Carnegie Mellon University, Pittsburgh, PA, USA
Jörg Siekmann, University of Saarland, Saarbrücken, Germany

Volume Editors

Chitta Baral
Arizona State University
Department of Computer Science and Engineering
Fulton School of Engineering
Brickyard Suite 572, 699 S. Mill Avenue, Tempe, AZ 85281, USA
E-mail: chitta@asu.edu

Gianluigi Greco
Nicola Leone
Giorgio Terracina
Università della Calabria
Dipartimento di Matematica
via P. Bucci 30B, I-87030 Rende (CS), Italy
E-mail: {ggreco, leone, terracina}@mat.unical.it

Library of Congress Control Number: 2005931127

CR Subject Classification (1998): I.2.3, I.2, F.4.1, D.1.6

ISSN 0302-9743
ISBN-10 3-540-28538-5 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-28538-0 Springer Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer. Violations are liable to prosecution under the German Copyright Law.

Springer is a part of Springer Science+Business Media

springeronline.com

© Springer-Verlag Berlin Heidelberg 2005
Printed in Germany

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 11546207 06/3142 5 4 3 2 1 0

Preface

These are the proceedings of the *8th International Conference on Logic Programming and Nonmonotonic Reasoning* (LPNMR 2005). Following the previous ones held in Washington, DC, USA (1991), Lisbon, Portugal (1993), Lexington, KY, USA (1995), Dagstuhl, Germany (1997), El Paso, TX, USA (1999), Vienna, Austria (2001) and Ft. Lauderdale, FL, USA (2004), the eighth conference was held in Diamante, Italy, from 5th to 8th of September 2005.

The aim of the LPNMR conferences is to bring together and facilitate interactions between active researchers interested in all aspects concerning declarative logic programming, nonmonotonic reasoning, knowledge representation, and the design of logic-based systems and database systems. LPNMR strives to encompass theoretical and experimental studies that lead to the implementation of practical systems for declarative programming and knowledge representation.

The technical program of LPNMR 2005 comprised three invited talks that were given by Jürgen Angele, Thomas Eiter and Michael Kifer. All papers presented at the conference and published in these proceedings went through a rigorous review process which selected 25 research papers and 16 papers for the system and application tracks.

Many individuals worked for the success of the conference. Special thanks are due to all members of the Program Committee and to additional reviewers for their efforts to produce fair and thorough evaluations of submitted papers. A special thanks is due to the University of Calabria Organizing Committee which made this event possible. Last, but not least, we thank the sponsoring institutions for their generosity.

June 2005

Chitta Baral and Nicola Leone
Program Co-chairs
LPNMR'05

Organization

LPNMR 2005 was organized by the Department of Mathematics at the University of Calabria, Italy.

Program Co-chairs

Chitta Baral (Arizona State University, USA)
Nicola Leone (University of Calabria, Italy)

Program Committee

Jose Alferes (New University of Lisbon, Portugal)
Leopoldo Bertossi (Carleton University, Canada)
Pedro Cabalar (Corunna University, Spain)
Gerhard Brewka (University of Leipzig, Germany)
Juergen Dix (Technical University of Clausthal, Germany)
Wolfgang Faber (University of Calabria, Italy)
Norman Foo (University of New South Wales, Australia)
Michael Gelfond (Texas Tech University, USA)
Antonis Kakas (University of Cyprus, Cyprus)
Katsumi Inoue (Kobe University, Japan)
Vladimir Lifschitz (University of Texas at Austin, USA)
Fangzhen Lin (Hong Kong University of Science and Technology, China)
Ilkka Niemelä (Helsinki University of Technology, Finland)
David Pearce (Rey Juan Carlos University, Spain)
Alessandro Provetti (University of Messina, Italy)
Francesco Scarcello (University of Calabria, Italy)
Torsten Schaub (University of Potsdam, Germany)
Hans Tompits (Vienna University of Technology, Austria)
Francesca Toni (Imperial College London, UK)
Mirek Truszczynski (University of Kentucky, USA)
Marina de Vos (University of Bath, United Kingdom)

Committee for the System and Application Tracks

Gerald Pfeifer (SUSE/Novell) - [**chair**]
Marcello Balduccini (Texas Tech University)
Yuliya Lierler (Universität Erlangen)
Ilkka Niemelä (Helsinki University of Technology)
Yuting Zhao (Istituto Trentino di Cultura)

Publicity Chair

Gianluigi Greco (University of Calabria, Italy)

Organizing Co-chairs

Giovambattista Ianni (University of Calabria, Italy)
Giorgio Terracina (University of Calabria, Italy)

Organizing Committee

Francesco Calimeri
Mina Catalano
Chiara Cumbo
Tina Dell'Armi
Stefania Galizia
Gianluigi Greco
Giuseppe Ielpa
Vincenzino Lio
Simona Perri
Veronica Policicchio
Francesco Ricca
Paola Sdao

External Referees

João Alcântara
Christian Anger
Marcello Balduccini
Federico Banti
Andrea Bracciali
Martin Brain
Loreto Bravo
Andreas Bruening
Francesco Buccafurri
Stefania Costantini
Carlos Damasio
Yannis Dimopoulos
Phan Minh Dung
Ulle Endriss
Esra Erdem
Paolo Ferraris
Michael Fink
Martin Gebser
Gianluigi Greco
Keijo Heljanko
Koji Iwanuma
Tomi Janhunen
Joohyung Lee
João Leite
Lengning Liu
Massimo Marchi
Victor Marek
Yves Martin
Thomas Meyer
Rob Miller
Emilia Oikarinen
Philip Reiser
Fariba Sadri
Chiaki Sakama
Ken Satoh
Juan Manuel Serrano
Tommi Syrjänen
Giorgio Terracina
Satoshi Tojo
Richard Watson
Gregory Wheeler
Stefan Woltran
Ester Zumpano

Table of Contents

Invited Papers

Nonmonotonic Reasoning in FLORA-2
Michael Kifer 1

Data Integration and Answer Set Programming
Thomas Eiter 13

Halo I: A Controlled Experiment for Large Scale Knowledge Base Development
Jürgen Angele, Eddie Moench, Henrik Oppermann, Dirk Wenke 26

ASP Foundations

Unfounded Sets for Disjunctive Logic Programs with Arbitrary Aggregates
Wolfgang Faber 40

Loops: Relevant or Redundant?
Martin Gebser, Torsten Schaub 53

Approximating Answer Sets of Unitary Lifschitz-Woo Programs
Victor W. Marek, Inna Pivkina, Mirosław Truszczyński 66

On Modular Translations and Strong Equivalence
Paolo Ferraris 79

ASP Extensions

Guarded Open Answer Set Programming
Stijn Heymans, Davy Van Nieuwenborgh, Dirk Vermeir 92

External Sources of Computation for Answer Set Solvers
Francesco Calimeri, Giovambattista Ianni 105

Answer Sets for Propositional Theories
Paolo Ferraris 119

Applications

An ID-Logic Formalization of the Composition of Autonomous Databases
Bert Van Nuffelen, Ofer Arieli, Alvaro Cortés-Calabuig, Maurice Bruynooghe 132

On the Local Closed-World Assumption of Data-Sources
Alvaro Cortés-Calabuig, Marc Denecker, Ofer Arieli, Bert Van Nuffelen, Maurice Bruynooghe 145

Computing Dialectical Trees Efficiently in Possibilistic Defeasible Logic Programming
Carlos I. Chesñevar, Guillermo R. Simari, Lluis Godo 158

Actions and Causations

An Approximation of Action Theories of $\mathcal{AL}$ and Its Application to Conformant Planning
Tran Cao Son, Phan Huy Tu, Michael Gelfond, A. Ricardo Morales 172

Game-Theoretic Reasoning About Actions in Nonmonotonic Causal Theories
Alberto Finzi, Thomas Lukasiewicz 185

Some Logical Properties of Nonmonotonic Causal Theories
Marek Sergot, Robert Craven 198

$\mathcal{M}$odular-$\mathcal{E}$: An Elaboration Tolerant Approach to the Ramification and Qualification Problems
Antonis Kakas, Loizos Michael, Rob Miller 211

Algorithms and Computation

PLATYPUS: A Platform for Distributed Answer Set Solving
Jean Gressmann, Tomi Janhunen, Robert E. Mercer, Torsten Schaub, Sven Thiele, Richard Tichy 227

Solving Hard ASP Programs Efficiently
Wolfgang Faber, Francesco Ricca 240

Mode-Directed Fixed Point Computation
Hai-Feng Guo 253

Lookahead in Smodels Compared to Local Consistencies in CSP
Jia-Huai You, Guohua Liu, Li Yan Yuan, Curtis Onuczko 266

Foundations

Nested Epistemic Logic Programs
Kewen Wang, Yan Zhang 279

An Algebraic Account of Modularity in ID-Logic
Joost Vennekens, Marc Denecker 291

Default Reasoning with Preference Within Only Knowing Logic
Iselin Engan, Tore Langholm, Espen H. Lian, Arild Waaler 304

Semantics

A Social Semantics for Multi-agent Systems
Francesco Buccafurri, Gianluca Caminiti 317

Revisiting the Semantics of Interval Probabilistic Logic Programs
Alex Dekhtyar, Michael I. Dekhtyar 330

Routley Semantics for Answer Sets
Sergei Odintsov, David Pearce 343

The Well Supported Semantics for Multidimensional Dynamic Logic Programs
Federico Banti, José Júlio Alferes, Antonio Brogi, Pascal Hitzler 356

Application Track

Application of Smodels in Quartet Based Phylogeny Construction
Gang Wu, Jia-Huai You, Guohui Lin 369

Using Answer Set Programming for a Decision Support System
Christoph Beierle, Oliver Dusso, Gabriele Kern-Isberner 374

Data Integration: A Challenging ASP Application
Nicola Leone, Thomas Eiter, Wolfgang Faber, Michael Fink, Georg Gottlob, Luigi Granata, Gianluigi Greco, Edyta Kałka, Giovambattista Ianni, Domenico Lembo, Maurizio Lenzerini, Vincenzino Lio, Bartosz Nowicki, Riccardo Rosati, Marco Ruzzi, Witold Staniszkis, Giorgio Terracina 379

Abduction and Preferences in Linguistics
Kathrin Konczak, Ralf Vogel 384

Inference of Gene Relations from Microarray Data by Abduction
Irene Papatheodorou, Antonis Kakas, Marek Sergot 389

System Track

nomore$^{<}$: A System for Computing Preferred Answer Sets
Susanne Grell, Kathrin Konczak, Torsten Schaub 394

Integrating an Answer Set Solver into Prolog: ASP – PROLOG
Omar Elkhatib, Enrico Pontelli, Tran Cao Son 399

CIRC2DLP — Translating Circumscription into Disjunctive Logic Programming
Emilia Oikarinen, Tomi Janhunen 405

Pbmodels — Software to Compute Stable Models by Pseudoboolean Solvers
Lengning Liu, Mirosław Truszczyński 410

KMONITOR – A Tool for Monitoring Plan Execution in Action Theories
Thomas Eiter, Michael Fink, Ján Senko 416

The *nomore++* System
Christian Anger, Martin Gebser, Thomas Linke, André Neumann, Torsten Schaub ... 422

SMODELS^{A} — A System for Computing Answer Sets of Logic Programs with Aggregates
Islam Elkabani, Enrico Pontelli, Tran Cao Son 427

A DLP System with Object-Oriented Features
Francesco Ricca, Nicola Leone, Valerio De Bonis, Tina Dell'Armi, Stefania Galizia, Giovanni Grasso ... 432

Testing Strong Equivalence of Datalog Programs - Implementation and Examples
Thomas Eiter, Wolfgang Faber, Patrick Traxler 437

SELP - A System for Studying Strong Equivalence Between Logic Programs
Yin Chen, Fangzhen Lin, Lei Li 442

CMODELS – SAT-Based Disjunctive Answer Set Solver
Yuliya Lierler 447

Author Index 453

Nonmonotonic Reasoning in FLORA-2*

Michael Kifer

Department of Computer Science,
State University of New Your at Stony Brook,
Stong Brook, NY 11794, USA
kifer@cs.stonybrook.edu

Abstract. FLORA-2 is an advanced knowledge representation system that integrates F-logic, HiLog, and Transaction Logic. In this paper we give an overview of the theoretical foundations of the system and of some of the aspects of nonmonotonic reasoning in FLORA-2. These include scoped default negation, behavioral inheritance, and nonmonotonicity that stems from database dynamics.

1 Introduction

FLORA-2 is a knowledge base engine and a complete environment for developing knowledge-intensive applications. It integrates F-logic with other novel formalisms such as HiLog and Transaction Logic. FLORA-2 is freely available on the Internet[1] and is in use by a growing research community. Many of the features of FLORA-2 have been adopted by the recently proposed languages in the Semantic Web Services domain: WSML-Rule[2] and SWSL-Rules.[3]

One of the main foundational ingredients of FLORA-2, F-logic [20], extends classical predicate calculus with the concepts of objects, classes, and types, which are adapted from object-oriented programming. In this way, F-logic integrates the paradigms of logic programming and deductive databases with the object-oriented programming paradigm. Most of the applications of F-logic have been in intelligent information systems, but more recently it has been used to represent ontologies and other forms of Semantic Web reasoning [14,12,27,1,11,2,19].

HiLog [8] is an extension of the standard predicate calculus with higher-order syntax. Yet the semantics of HiLog remains first-order and tractable. In FLORA-2, HiLog is the basis for simple and natural querying of term structures and for reification (or objectification) of logical statements, which is an important requirement for a Semantic Web language. Transaction Logic [6] provides the basis for declarative programming of "procedural knowledge" that is often embedded in intelligent agents or Semantic Web services.

In this paper we first survey the main features of FLORA-2 and then discuss three forms of nonmonotonic reasoning provided by the system.

* This work was supported in part by NSF grant CCR-0311512 and by U.S. Army Medical Research Institute under a subcontract through Brookhaven National Lab.

1 http://flora.sourceforge.net/

2 http://www.w3.org/Submission/WSML/

3 http://www.daml.org/services/swsl-rules/1.0/

C. Baral et al. (Eds.): LPNMR 2005, LNAI 3662, pp. 1–12, 2005.
© Springer-Verlag Berlin Heidelberg 2005

2 Overview of F-Logic

F-logic extends predicate calculus both syntactically and semantically. It has a monotonic logical entailment relationship, and its proof theory is sound and complete with respect to the semantics. F-logic comes in two flavors: the first-order flavor and the logic programming flavor. The first-order flavor of F-logic can be viewed as a syntactic variant of classical logic [20]. The logic programming flavor uses a subset of the syntax of F-logic, but gives it a different, non-first-order semantics by interpreting the negation operator as negation-as-failure.

The relationship between the first-order variant of F-logic and its logic programming variant is similar to the relationship between predicate calculus and standard logic programming [23]: object-oriented logic programming is built on the rule-based subset of F-logic by adding the appropriate non-monotonic extensions [32,33,24]. These extensions are intended to capture the semantics of negation-as-failure (like in standard logic programming [28]) and the semantics of multiple inheritance with overriding (which is not found in standard logic programming).

F-logic uses first-order variable-free terms to represent *object identity* (abbr., OID); for instance, `John` and `father(Mary)` are possible Ids of objects. Objects can have attributes. For instance,

```
Mary[spouse -> John, children -> {Alice,Nancy}].
Mary[children -> Jack].
```

Such formulas are called F-logic *molecules*. The first formula says that object `Mary` has an attribute `spouse` whose value is the OID `John`. It also says that the attribute `children` is set-valued and its value is a set that *contains* two OIDs: `Alice` and `Nancy`. We emphasize "contains" because sets do not need to be specified all at once. For instance, the second formula above says that `Mary` has an additional child, `Jack`.

In earlier versions of F-logic, set-valued attributes were denoted with `->>` instead of `->`. However, subsequently the syntax was modernized and simplified. Instead of using different arrows, cardinality constraints (to be explained shortly) were introduced to indicate that an attribute is single-valued.

While some attributes of an object are specified explicitly, as facts, other attributes can be defined using deductive rules. For instance, we can derive `John[children -> {Alice,Nancy,Jack}]` using the following deductive rule:

```
?X[children -> {?C}] :- ?Y[spouse -> ?X, children -> {?C}].
```

In the new and simplified syntax, alphanumeric symbols prefixed with the ?-sign denote variables and unprefixed alphanumeric symbols denote constants (i.e., OIDs). The earlier versions of FLORA-2 used Prolog conventions where variables were capitalized alphanumeric symbols.

F-logic objects can also have *methods*, which are functions that take arguments. For instance,

```
John[grade(cs305,fall2004) -> 100, courses(fall2004) -> {cs305,cs306}].
```

says that `John` has a method, `grade`, whose value on the arguments `cs305` (a course identifier) and `fall2004` (a semester designation) is `100`; it also has a set-valued method `courses`, whose value on the argument `fall2004` is a set of OIDs that contains course identifiers `cs305` and `cs306`. Like attributes, methods can be defined using deductive rules.

The F-logic syntax for *class membership* is `John:student` and for *subclass relationship* it is `student::person`. Classes are treated as objects and it is possible for the same object to play the role of a class in one formula and of an object in another. For instance, in the formula `student:class`, the symbol `student` plays the role of an object, while in `student::person` it appears in the role of a class.

F-logic also provides means for specifying schema information through *signature* formulas. For instance, `person[spouse {0:1}=>person, name {0:1}=> string, child=> person]` is a signature formula that says that class `person` has three attributes: single-valued attributes `spouse` and `name` (single-valuedness is indicated by the cardinality constraint `0:1`) and a set-valued attribute `child`. It further says that the first attribute returns objects of type `person`, the second of type `string`, and the last returns sets of objects such that each object in the set is of type `person`.

3 HiLog and Meta-information

F-logic provides simple and natural means for exploring the structure of object data. Both the schema information associated with classes and the structure of individual objects can be queried by simply putting variables in the appropriate syntactic positions. For instance, to find the set-valued methods that are defined in the *schema* of class `student` and return objects of type `person`, one can ask the following query:

```
?- student[?M=>person].
```

The next query is about the type of the results of the attribute `name` in class `student`. This query also returns all the superclasses of class `student`.

```
?- student::?C and student[name=>?T].
```

The above are schema-level meta-queries: they involve the subclass relationship and the type information. One can also pose meta-queries that involve object data (rather than schema). The following queries return the methods that have a known value for the object `John`:

```
?- John[?Meth->?SomeValue].
```

However, the meta-query facilities of F-logic are not complete. For instance, there is no way in such queries to separate method names from their arguments. Thus, if we had a fact of the form

```
John[age(2005)-> 20].
```

then the first of the above queries will bind ?Meth to age(2005)—we cannot separate age from 2005.

This is where HiLog [8] comes into picture. In HiLog, second-order syntax is allowed and so variables can appear in positions of function and predicate symbols. For instance, queries such as

```
?- person[?M(?Arg)->?SomeValue].
?- person[?M(?Arg)=>integer].
```

are allowed and ?M would be bound to age and, possibly, to other values as well. The semantics for this second-order syntax is first order, however. Roughly it means that variables get bound not to the extensional values of the symbols (i.e., the actual relations that are used to interpret the function and predicate symbols), but to the symbols themselves. Details of these semantics can be found in [8].

HiLog does not stop at allowing variables over function and predicate symbols—it also permits them over atomic formulas. For instance, the following query is legal and will succeed with ?X bound to p(a).

```
p(a).
q(p(a)).
?- q(?X), ?X.
```

What happens here is that the proposition p(a) is *reified* (made into an object) and so it can be bound to a variable. HiLog's reification of atomic formulas can be extended to arbitrary quantifier-free formula of the rule-based subset of HiLog and F-logic, and this has been done in [31,19]. For instance, one can say that John believes that Mary likes Sally as follows:

$$\mathtt{John}[\mathtt{believes}\,\mathtt{->}\,\$\{\mathtt{Mary}[\mathtt{likes}\,\mathtt{->}\,\mathtt{Sally}]\}]. \tag{1}$$

Here ${...} is the syntax that FLORA-2 uses to denote reified statements. An example of a more complicated reified statement is this:

$$\mathtt{John}[\mathtt{believes}\,\mathtt{->}\,\$\{\mathtt{Bob}[\mathtt{likes}\,\mathtt{->}\,?\mathtt{X}]\;:-\;\mathtt{Mary}[\mathtt{likes}\,\mathtt{->}\,?\mathtt{X}]\}]. \tag{2}$$

This sentence reifies a rule (not just a fact) and states that John also believes that Bob likes anybody who is liked by Mary. Combined with the previous statement that John believes that Mary likes Sally, one would expect that John would also believe that Bob likes Sally. However, we cannot conclude this just yet because we do not know that John is a rational being who applies modus ponens in his daily life. But this rational principle can be stated rather easily:

$$\begin{aligned}&\mathtt{John}[\mathtt{believes}\;\mathtt{->}\;?\mathtt{A}]\;:-\\&\qquad\qquad\mathtt{John}[\mathtt{believes}\;\mathtt{->}\;\{\$\{?\mathtt{Head}\;:-\;?\mathtt{Body}\},\;?\mathtt{Body}\}].\end{aligned} \tag{3}$$

4 Transaction Logic

Knowledge intensive applications, such as those in semantic Web services and intelligent agents, often require primitives for modifying the underlying state

of the system. Prolog provides the well-known `assert` and `retract` operators, which are non-logical and are therefore widely viewed as problematic. Various attempts to formalize updates in a logic programming language have had only a limited success (*e.g.*, [21,26,22]). A detailed discussion of this subject appears in [5,6]. Some of the most serious problems with these approaches is that they impose special programming styles (which is a significant burden) and that they do not support subroutines — one of the most fundamental aspects of any programming language.

Transaction Logic [4,5,6] is a comprehensive solution to the problem of updates in logic programming. This solution has none of the above drawbacks and it fits nicely with the traditional theory of logic programming. The use of Transaction Logic has been illustrated on a vast variety of applications, ranging from databases to robot action planning to reasoning about actions to workflow analysis and Web services [5,7,10,19].

An important aspect of the update semantics of Transaction Logic is that updates are *atomic*, which means that an update transaction executes in its entirety or not at all. In contrast, in Prolog, if a post-condition of a state-changing predicate is false, the execution "fails" but the changes made by `assert` and `retract` would stay and the knowledge base is left in a inconsistent state. This property is responsible for many complications in Prolog programming. This and related problems are rectified by Transaction Logic semantics.

FLORA-2 integrates F-logic and Transaction Logic along the lines of [18] with certain refinements that distinguish queries from transactions and thus enable a number of compile-time checks. In Transaction Logic, both actions (transactions) and queries are represented as predicates. In FLORA-2, transactions are expressed as object methods that are prefixed with the special symbol "%".

The following program is an implementation of a block-stacking robot in FLORA-2. Here, the action `stack` is defined as a Boolean method of the robot.

```
?R[%stack(0,?X)]  :- ?R:robot.
?R[%stack(?N,?X)] :- ?R:robot, ?N > 0,
                     ?Y[%move(?X)], ?R[%stack(?N-1,?Y)].
?Y[%move(?X)]     :- ?Y:block, ?Y[clear], ?X[clear], ?X[widerThen(?Y)],
                     btdelete{?Y[on->?Z]}, btinsert{?Z[clear]},
                     btinsert{?Y[on->?X]}, btdelete{?X[clear]}.
```

The primitives `btdelete` and `btinsert` are FLORA-2's implementations of the insert and delete operators with the Transaction Logic semantics. Informally, the above rules say that to stack a pyramid of `N` blocks on top of block `?X`, the robot must find a block `?Y`, move it onto `?X`, and then stack `N-1` blocks on top of `?Y`. To move `?Y` onto `?X`, both blocks must be "clear" (*i.e.*, with no other block sitting on top of them), and `?X` must be wider than `?Y`. If these conditions are met, the database will be updated accordingly. If any of the conditions fails, it means that the current attempted execution is not a valid try and another attempt will be made. If no valid execution is found, the transaction fails and no changes will be made to the database.

A simple-minded translation of the above program into Prolog is incorrect, since such a program might leave the database in an inconsistent state. A *correct* version of the above FLORA-2 program in Prolog is more complicated and much less natural.

5 Scoped Default Negation

Closed world assumption [25] is an inference rule for negative information. It states that in the absence of a positive proof that a fact, `F`, is true one must conclude that `not F` is true. Negation that obeys such an inference rule is *not* classical and is often called *default negation*.[4]

Various forms of the closed-world assumption (CWA) have been successfully used in database and logic programing applications for over thirty years now and vast experience has been accumulated with the use of this paradigm in knowledge representation [9,28,16]. In contrast, classical logic is based on the open-world assumption (OWA), and this has been the *sine qua non* in, for example, the description logic community. Each community allowed the other to continue to believe in its respective heresy until the Semantic Web came along.

The advent of the Semantic Web caused heated discussions about the one and only kind of negation that is suitable for this emerging global knowledge base (see, e.g., `http://robustai.net/papers/Monotonic_Reasoning_on_the _Semantic _Web. html` for a compendium). The main argument against closed-world assumption goes like this. The Web is practically infinite and failure to derive some fact from the currently obtained information does not warrant the conclusion that this fact is false. Nevertheless, thirty years of experience in practical knowledge representation cannot be dismissed lightly and even the proponents of the open-world assumption are beginning to realize that. One idea that is beginning to take hold is that CWA is acceptable—even in the Web environment—as long as the *scope* of the closure is made explicit and concrete [17]. A simplified form of this idea was recently added to the N3 rule language [3] (whose author, Tim Berners-Lee, previously resisted the use of default negation).

Scoped default negation was introduced in FLORA-2 as part of its innovative architecture for knowledge base modules. It is related to (but is different from) the so called *local closed world assumption* [13]. In FLORA-2, a module is a container for a concrete knowledge base (or a part of it). Modules isolate the different parts of a knowledge base and provide a clean interface by which these parts can interact. Modules can be created dynamically, associated with knowledge bases on the fly, and they support a very powerful form of encapsulation. For our discussion, the relevant aspect of the FLORA-2 modules is that they provide a simple and natural mechanism for scoped default negation.

Consider the statements (1), (2), and (3) about John's beliefs from the end of Section 3. To use these statements, one must insert them into a module,

[4] Some researchers also sometimes call this type of negation *negation-as-failure*. We avoid this terminology because negation-as-failure was originally used to denote a specific proof strategy used in the Prolog language.

let us call it johnmodule, by, for instance, loading the file that contains these statements into the module. To query the information about John's beliefs one would now pose queries such as

```
?- John[believes -> ${Mary[likes->Sally]}]@johnmodule.
```

Similarly, to inquire whether John *does not* believe that Bob is Sally's husband one would ask the query

```
?- not John[believes->${Sally[spouse->Bob]}]@johnmodule.        (4)
```

The scope of the above query is limited to the module johnmodule only. If it cannot be derived that John[believes->${Sally[spouse->Bob]}] is true from the knowledge base residing in the module then (and only then) the answer will be "Yes." The answer to (4) will remain the same even if some other module asserts that John[believes->${Sally[spouse->Bob]}] because the scope of the default negation in the query is limited to the module johnmodule. It is, however, possible to ask unrestricted negative queries by placing a variable in the module position:

```
?- not John[believes -> ${Sally[spouse->Bob]}]@?Mod.
```

This query returns "Yes" iff John is not known to believe that Sally[spouse-> Bob] is true in every module (that is registered with the system).

The semantics of FLORA-2 modules is very simple. The attribute and method names of the formulas that are loaded into a module, such as john module, are uniquified so that the same attribute name in the program will be given different and unique *real* names in different modules. For instance, a formula such as John[believes->abc] might become John[believes#foo->abc] in module foo and John[believes#bar->abc] in module bar. Due to this transformation, the query (4) turns into the following query in the actual knowledge base:

```
?- not John[believes#johnmodule-> ${Sally[spouse#johnmodule->Bob]}].
```

Since other modules cannot have facts or rules whose heads have the form ...[believes#johnmodule->...], the answer "Yes" or "No" depends only on the information stored in module johnmodule.

6 Nonmonotonic Inheritance

F-logic supports both *structural* and *behavioral* inheritance. The former refers to inheritance of method types from superclasses to their subclasses and the latter deals with inheritance of method definitions from superclasses to subclasses.

Structural inheritance is defined by very simple inference rules:

If subcl::cl, cl[attr*=>type] **then** subcl[attr*=>type]
If obj:cl, cl[attr*=>type] **then** obj[attr=>type]

The statement `cl[attr *=> type]` above says that `attr` is an *inheritable* attribute, which means that both its type and value are inheritable by the subclasses and members of class `cl`. Inheritability of the type of an attribute is indicated with the star attached to the arrow: `*=>`. In all of our previous examples we have been dealing with *non-inheritable* attributes, which were designated with star-less arrows. Note that when the type of an attribute is inherited to a subclass it remains inheritable. However, when it is inherited to a member of the class it is no longer inheritable.

Type inheritance, as defined by the above rules, is *monotonic* and thus is peripheral to the subject of this paper. Behavioral inheritance is more complex. To get a flavour of behavioral inheritance, consider the following knowledge base:

```
royalElephant::elephant.
clyde:royalElephant.
elephant[color *-> grey].
royalElephant[color *-> white].
```

As with type definitions, a star attached to the arrow `*->` indicates inheritability. For instance, `color` is an inheritable attribute in classes `elephant` and `royalElephant`. The inference rule that guides behavioral inheritance can informally be stated as follows. If `obj` is an object and `cl` is a class, then

`obj:cl, cl[attr *-> value]`   should imply   `obj[attr -> value]`

unless the inheritance is overwritten by a more specific class. The meaning of the exception here is that the knowledge base should *not* imply the formula `obj[attr -> value]` if there is an intermediate class, `cl′`, which overrides the inheritance, i.e., if `obj` : `cl′`, `cl′` :: `cl` are true and `cl′`[`attr *-> value′`] (for some `value′` $\neq$ `value`) is defined explicitly.[5] A similar exception exists in case of multiple inheritance conflicts. Note that inheritable attributes become non-inheritable after they are inherited by class members. In the above case, inheritance of the color *grey* is overwritten by the color *white* and so `clyde[color -> white]` is derived by the rule of inheritance.

This type of inheritance is clearly nonmonotonic. For instance, if in the above example we add the fact `clyde[color->yellow]` to the knowledge base then `clyde[color->white]` is no longer inferred by inheritance (the inference is said to be overwritten).

Model-theoretic semantics for nonmonotonic inference by inheritance is rather subtle and has eluded researchers for many years. Although the above informal rules for inference by inheritance seem natural, there are subtle problems when behavioral inheritance is used together with deductive rules. To understand the problem, consider the following example:

```
cl[attr *-> v1].
subcl::cl.
```

[5] The notion of an explicit definition seems obvious at first but, in fact, is quite subtle. Details can be found in [29].

```
obj:subcl.
subcl[attr*=>v2] :- obj[attr->v1].
```

If we apply the rule of inheritance to this knowledge base, then `obj[attr->v1]` should be inherited, since no overriding takes place. However, once `obj[attr-> v1]` is derived by inheritance, `subcl[attr*=>v2]` can be derived by deduction—and now we have a chicken-and-egg problem. Since `subcl` is a more specific superclass of `obj`, the derivation of `subcl[attr*=>v2]` appears to override the earlier inheritance of `obj[attr->v1]`. But this, in turn, undermines the very reason for deriving `subcl[attr*=>v2]`. The above is only one of several suspicious derivation patterns that arise due to interaction of inheritance and deduction. The original solution reported in [20] was not model-theoretic and was problematic in several other respects as well. A satisfactory and completely model-theoretic solution was proposed in [29,30].

7 Database Dynamics and Nonmonotonicity

In [5], a generalization of the perfect-model semantics was defined for Transaction Logic programs with negation in the rule body. This semantics was implemented in FLORA-2 only partially, with negation applicable only to non-transactional formulas in the rule body. For instance, the following transaction logs all unauthorised accesses to any given resource, and default negation, `not`, is applied only to a query (not an action that has a side effect):

$$\begin{aligned} &\texttt{?Rsrc}[\%\texttt{recordUnauthAaccess}(\texttt{?Agent})] :- \\ &\qquad \texttt{not ?Rsrc}[\texttt{eligible->?Agent}], \\ &\qquad \texttt{insert}\{\texttt{unAuthLog}(\texttt{?Agent}, \texttt{?Rsrc})\}. \end{aligned} \tag{5}$$

Nonmonotonicity comes into play here in a somewhat different sense than in standard logic programming (ignoring the non-logical `assert` and `retract`). In the standard case, nonmonotonicity means that certain formulas that were derivable in a database state, `s1`, will not be derivable in the state obtained from `s1` by adding more facts. In the above example, however, no inference that was enabled by rule (5) can become invalidated by adding more facts, since this rule is not a statement about the initial database state.

In our example, nonmonotonicity reveals itself in a different way: the transaction of the form `?- mySecrets[%recordUnauthAaccess(John)]` can be executable in the initial state (if `mySecrets[eligible->John]` is not derivable) and non-executable in the state obtained by adding `mySecrets[eligible->John]` to the original state.

This kind of non-monotonicity can be stated formally in the logic as: there are database states $\mathbf{D}$ and $\mathbf{D}'$, where $\mathbf{D} \subseteq \mathbf{D}'$, such that

$$\mathbf{D}\texttt{---} \models \texttt{mySecrets}[\%\texttt{recordUnauthAaccess}(\texttt{John})]$$

but

$$\mathbf{D}' \models \texttt{not} \Diamond \texttt{mySecrets}[\%\texttt{recordUnauthAaccess}(\texttt{John})]$$

The first statement above is called *executional entailment*; it means that there is a sequence of states, beginning with the given state **D**, which represents an execution path of the transaction `mySecrets[%recordUnauthAaccess(John)]`. The second statement says that there is no execution path, which starts at state $\mathbf{D}'$, for the transaction `mySecrets[%recordUnauthAaccess(John)]`.

Another instance of nonmonotonic behavior that is different from the classical cases occurs when enlarging the initial state of transaction execution leads to a possible elimination of facts in the final state of transaction execution. To illustrate this, consider a slightly modified version of transaction (5):

```
?Rsrc[%recordUnauthAaccess(?Agent)] :-
      not ?Rsrc[eligible->?Agent], insert{unAuthLog(?Agent,?Rsrc)}.
?Rsrc[%recordUnauthAaccess(?Agent)] :- ?Rsrc[eligible->?Agent].
```

In this case, the transaction `?- mySecrets[%recordUnauthAaccess(John)]` can be executed regardless of whether John is eligible or not. If John is not eligible then `unAuthLog(John,mySecrets)` becomes true in the final state of the execution of this transaction. If John is already eligible then nothing changes. Now, if we add the fact that John *is* eligible to access `mySecrets` then the transaction executes without changing the state. Therefore, `unAuthLog(John,mySecrets)` is no longer derivable in the final state. Thus, enlarging the initial state of transaction execution does not necessarily lead to a monotonic enlargement of the final state.

8 Conclusion

This paper presents an overview of the formal foundations of the FLORA-2 system with a focus on various forms of nonmonotonic reasoning in the system. Three aspects have been considered: scoped default negation, behavioral inheritance, and nonmonotonicity that stems from database dynamics. Scoped negation is believed to be the right kind of negation for the Semantic Web. Behavioral inheritance is an important concept in object-oriented modeling; in FLORA-2 it has been extended to work correctly (from the semantic point of view) in a rule-based system. Finally, we discussed database dynamics in FLORA-2 and have shown how it can lead to of nonmonotonic behavior.

References

1. J. Angele and G. Lausen. Ontologies in F-logic. In S. Staab and R. Studer, editors, *Handbook on Ontologies in Information Systems*, pages 29–50. Springer Verlag, Berlin, Germany, 2004.
2. D. Berardi, H. Boley, B. Grosof, M. Gruninger, R. Hull, M. Kifer, D. Martin, S. McIlraith, J. Su, and S. Tabet. SWSL: Semantic Web Services Language. Technical report, Semantic Web Services Initiative, April 2005. http://www.daml.org/services/swsl/.
3. T. Berners-Lee. Primer: Getting into RDF & Semantic Web using N3, 2004. http://www.w3.org/2000/10/swap/Primer.html.

4. A. Bonner and M. Kifer. An overview of transaction logic. *Theoretical Comput. Sci.*, 133:205–265, October 1994.
5. A. Bonner and M. Kifer. Transaction logic programming (or a logic of declarative and procedural knowledge). Technical Report CSRI-323, University of Toronto, November 1995. `http://www.cs.toronto.edu/~bonner/transaction-logic.html`.
6. A. Bonner and M. Kifer. A logic for programming database transactions. In J. Chomicki and G. Saake, editors, *Logics for Databases and Information Systems*, chapter 5, pages 117–166. Kluwer Academic Publishers, March 1998.
7. A. Bonner and M. Kifer. Results on reasoning about action in transaction logic. In [15]. Springer-Verlag, 1998.
8. W. Chen, M. Kifer, and D. Warren. HiLog: A foundation for higher-order logic programming. *Journal of Logic Programming*, 15(3):187–230, February 1993.
9. K. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 292–322. Plenum Press, 1978.
10. H. Davulcu, M. Kifer, C. Ramakrishnan, and I. Ramakrishnan. Logic based modeling and analysis of workflows. In *ACM Symposium on Principles of Database Systems*, pages 25–33, Seattle, Washington, June 1998.
11. J. de Bruijn, H. Lausen, R. Krummenacher, A. Polleres, L. Predoiu, and D. Fensel. The WSML family of representation languages. Technical report, DERI, March 2005. http://www.wsmo.org/TR/d16/d16.1/.
12. S. Decker, D. Brickley, J. Saarela, and J. Angele. A query and inference service for RDF. In *QL'98 - The Query Languages Workshop*, December 1998.
13. O. Etzioni, K. Golden, and D. Weld. Sound and effcient closed-world reasoning for planning Artificial Intelligence. *Artificial Intelligence*, 89(1-2):113–148, 1997.
14. D. Fensel, M. Erdmann, and R. Studer. OntoBroker: How to make the WWW intelligent. In *Proceedings of the 11th Banff Knowledge Acquisition for Knowledge-Based Systems Workshop*, Banff, Canada, 1998.
15. B. Freitag, H. Decker, M. Kifer, and A. Voronkov, editors. *Transactions and Change in Logic Databases*, volume 1472 of *LNCS*. Springer-Verlag, Berlin, 1998.
16. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Logic Programming: Proceedings of the Fifth Conference and Symposium*, pages 1070–1080, 1988.
17. S. Hawke, S. Tabet, and C. de Sainte Marie. Rule Language Standardization: Report from the W3C Workshop on Rule Languages for Interoperability, May 2005. http://www.w3.org/2004/12/rules-ws/report/.
18. M. Kifer. Deductive and object-oriented data languages: A quest for integration. In *Int'l Conference on Deductive and Object-Oriented Databases*, volume 1013 of *Lecture Notes in Computer Science*, pages 187–212, Singapore, December 1995. Springer-Verlag. Keynote address at the 3d Int'l Conference on Deductive and Object-Oriented databases.
19. M. Kifer, R. Lara, A. Polleres, and C. Zhao. A logical framework for web service discovery. In *ISWC 2004 Semantic Web Services Workshop*. CEUR Workshop Proceedings, November 2004.
20. M. Kifer, G. Lausen, and J. Wu. Logical foundations of object-oriented and frame-based languages. *Journal of ACM*, 42:741–843, July 1995.
21. R. Kowalski. Database updates in event calculus. *Journal of Logic Programming*, 12(1&2):121–146, January 1992.
22. G. Lausen and B. Ludäscher. Updates by reasoning about states. In *2-nd International East/West Database Workshop*, Klagenfurt, Austria, September 1994.

23. J. W. Lloyd. *Foundations of Logic Programming (Second, extended edition).* Springer series in symbolic computation. Springer-Verlag, New York, 1987.
24. Ontoprise, GmbH. OntoBroker Manual. http://www.ontoprise.com/.
25. R. Reiter. On closed world databases. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 55–76. Plenum Press, New York, 1978.
26. R. Reiter. Formalizing database evolution in the situation calculus. In *Conference on Fifth Generation Computer Systems*, 1992.
27. S. Staab and A. Maedche. Knowledge portals: Ontologies at work. *The AI Magazine*, 22(2):63–75, 2000.
28. A. Van Gelder, K. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *J. ACM*, 38(3):620–650, July 1991.
29. G. Yang and M. Kifer. Well-founded optimism: Inheritance in frame-based knowledge bases. In *Intl. Conference on Ontologies, DataBases, and Applications of Semantics for Large Scale Information Systems (ODBASE)*, October 2002.
30. G. Yang and M. Kifer. Inheritance and rules in object-oriented semantic Web languages. In *Rules and Rule Markup Languages for the Semantic Web (RuleML03)*, volume 2876 of *Lecture Notes in Computer Science*. Springer Verlag, November 2003.
31. G. Yang and M. Kifer. Reasoning about anonymous resources and meta statements on the Semantic Web. *Journal on Data Semantics, LNCS 2800*, 1:69–98, September 2003.
32. G. Yang, M. Kifer, and C. Zhao. FLORA-2: A rule-based knowledge representation and inference infrastructure for the Semantic Web. In *International Conference on Ontologies, Databases and Applications of Semantics (ODBASE-2003)*, November 2003.
33. G. Yang, M. Kifer, and C. Zhao. FLORA-2: User's Manual. `http://flora.sourceforge.net/documentation.php`, March 2005.

Data Integration and Answer Set Programming

Thomas Eiter

Knowledge Based Systems Group, Institute of Information Systems,
Vienna University of Technology, A-1040 Vienna, Austria
eiter@kr.tuwien.ac.at

Abstract. The rapid expansion of the Internet and World Wide Web led to growing interest in data and information integration, which should be capable to deal with inconsistent and incomplete data. Answer Set solvers have been considered as a tool for data integration systems by different authors. We discuss why data integration can be an interesting model application of Answer Set programming, reviewing valuable features of non-monotonic logic programs in this respect, and emphasizing the role of the application for driving research.

1 Introduction

Triggered by the rapid expansion of the Internet and the World Wide Web, the integration of data and information from different sources has emerged as a crucial issue in many application domains, including distributed databases, cooperative information systems, data warehousing, or on-demand computing.

However, the problem is complex, and no canonical solution exists. Commercial software solutions such as IBM's Information Integrator [1] and academic systems (see e.g. [2]) fulfill only partially the ambitious goal of integrating information in complex application scenarios. In particular, handling inconsistent and/or incomplete data is, both semantically and computationally, a difficult issue, and is still an active area of research; for a survey of query answering on inconsistent databases, see [3].

In recent years, there has been growing interest in using non-monotonic logic programs, most prominently answer set solvers like DLV [4], Smodels [5], or Cmodels-2 [6] as a tool for data integration, and in particular to reconcile data inconsistency and incompleteness, e.g. [7,8,9,10,11,12,13,14,15]. In our opinion, data integration can in fact be viewed as an interesting model application of Answer Set Programming (ASP), for a number of different reasons:

1. The problem is important. There is rapidly growing interest in data and information integration, and this was estimated to be a $10 Billion market by 2006 [16].
2. Some of the key features of non-monotonic logic programming and ASP in particular, namely declarativity, expressiveness, and capability of nondeterminism can be fruitfully exploited.
3. Interest in ASP engines as a tool for solving data integration tasks emerged with people outside the ASP community, and in fact by different groups [7,8,11,12,13].
4. The application has raised new research problems and challenges, which have driven research to enhance and improve current ASP technology.

C. Baral et al (Eds.): LPNMR 2005, LNAI 3662, pp. 13–25, 2005.
© Springer-Verlag Berlin Heidelberg 2005

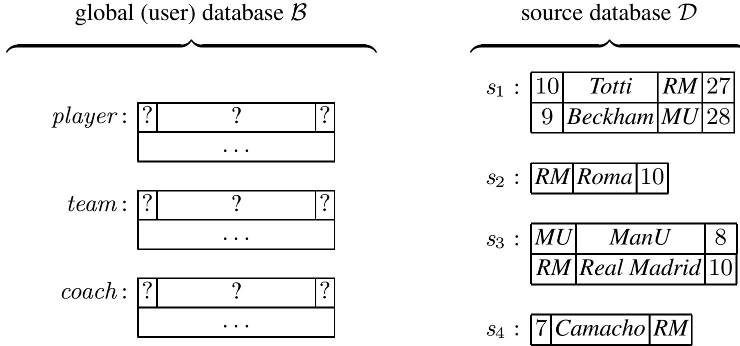


Fig. 1. Simple soccer data integration scenario – global and source relations

Example 1. To illustrate the problem, we consider a simple scenario of a data integration system which provides information about soccer teams. At the global (user) level, there are three relations

$player(Pcode, Pname, Pteam)$, $team(Tcode, Tname, Tleader)$, and
$coach(Ccode, Cname, Cteam)$,

which are interrelated with source relations

$s_1(A1, A2, A3, A4)$, $s_2(B1, B2, B3)$, $s_3(C1, C2, C3)$, and $s_4(D1, D2, D3)$

in the following way:

- $player$ correlates with the projection of s_1 to first three attributes;
- $team$ correlates with the union of s_2 and s_3; and
- $coach$ correlates with s_4.

(The precise form of correlation will be detailed later.) Now given the instance of the source database shown in Figure 1, how should a corresponding global database instance look like? In particular, if there are key constraints for the user relations, and further constraints like that a coach can neither be a player nor a team leader. Furthermore, if we want to pose a query which retrieves all players from the global relations (in logic programming terms, evaluate the rules

$$q(X) \leftarrow player(X, Y, Z)$$
$$q(X) \leftarrow team(V, W, X)$$

where q is a query predicate), how do we semantically determine the answer? □

Different approaches to data integration have been considered; see [17,2] for discussion. The most prominent ones are the *Global As View approach (GAV)*, in which the relations at the user level are amount to database views over the sources, and *Local As View approach (GAV)*, in which conversely the relations of the source database are

database views on the global relations. For both approaches, the usage of ASP has been explored, cf. [7,8,10,11,12,13,15].

In this remainder of this paper, we shall first briefly present a framework for data integration from the literature [17,18] which accommodates both GAV and LAV, along with proposals for semantics to deal with data inconsistencies. We then discuss why employing non-monotonic logic programs for this application is attractive, but also what shortcomings of ASP technology have been recognized, which have been driving (and still do so) research to improve ASP technology in order to meet the needs of this application. This is further detailed on the example of the INFOMIX information integration project, in which ASP has been adopted as the core computational technology to deal with data inconsistencies. We conclude with some remarks and issues for future research.

2 Data Integration Systems

While semi-structured data formats and in particular XML are gaining more and more importance in the database world, most of the theoretical work on advanced data integration has considered traditional relational databases, in which a database schema is modeled as a pair $\langle \Psi, \Sigma \rangle$ of a set Ψ of database relations and a set Σ of integrity constraints on them. The latter are first-order sentences on Ψ and the underlying (finite or infinite) set of constants (elementary values) *Dom*. A database instance can be viewed as a finite set of ground facts on Ψ and *Dom*, and is legal if it satisfies Σ.

A commonly adopted high-level structure of a data integration system $\mathcal{I}$ in a relational setting is a triple $\langle \mathcal{G}, \mathcal{S}, \mathcal{M} \rangle$ with the following components [17,18]:

1. $\mathcal{G} = \langle \Psi, \Sigma \rangle$ is a relational schema called the *global schema*, which represents the user's view of the integrated data. The integrity constraints Σ are usually from particular constraint classes, since the interaction of constraints can make semantic integration of data undecidable. Important classes of constraints are key constraints and functional dependencies (as well-known from any database course), inclusion dependencies (which enforce presence of certain tuples across relations), and exclusion dependencies (which forbid joint presence of certain tuples).
2. $\mathcal{S}$ is the *source schema*, which is given by the schemas of the various sources that are part of the data integration system. Assuming that they have been standardized apart, we can view $\mathcal{S}$ as a relational schema of the form $\mathcal{S} = \langle \Psi', \Sigma' \rangle$. The common setting is that Σ' is assumed to be empty, since the sources are autonomous and schema information may be not disclosed to the integration system.
3. $\mathcal{M}$ is the *mapping*, which establishes the relationship between $\mathcal{G}$ and $\mathcal{S}$ in a semantic way. The mapping consists of a collection of *mapping assertions* of the forms

$$(1) \quad q_{\mathcal{G}}(\boldsymbol{x}) \sqsubseteq q_{\mathcal{S}}(\boldsymbol{x}) \quad \text{and (2)} \quad q_{\mathcal{S}}(\boldsymbol{x}) \sqsubseteq q_{\mathcal{G}}(\boldsymbol{x}),$$

   where $q_{\mathcal{G}}(\boldsymbol{x})$ and $q_{\mathcal{S}}(\boldsymbol{x})$ are database queries (typically, expressible in first-order logic) with the same free variables $\boldsymbol{x}$, on $\mathcal{G}$ respectively $\mathcal{S}$.

In the above framework, the GAV and LAV approach result as special cases by restricting the queries $q_{\mathcal{G}}(\boldsymbol{x})$ respectively $q_{\mathcal{S}}(\boldsymbol{x})$ to atoms $r(\boldsymbol{x})$.

Example 2. (cont'd) Our running example scenario is represented as a data integration system $\mathcal{I} = \langle \mathcal{G}, \mathcal{S}, \mathcal{M} \rangle$, where $\mathcal{G}$ consists of the relations $player$, $team$, and $coach$. The associated constraints Σ are that the keys of $player$, $team$, and $coach$ are the attributes $\{Pcode, Pteam\}$, $\{Tcode\}$, and $\{Ccode, Cteam\}$, respectively, and that a coach can neither be a player nor a team leader. The source schema $\mathcal{S}$ comprises the relations s_1, s_2, s_3 and s_4. Finally, the GAV mapping $\mathcal{M}$ is defined in logic programming terms as follows (where $q_{\mathcal{S}} \sqsubseteq q_{\mathcal{G}}$ amounts to $q_{\mathcal{G}} \leftarrow q_{\mathcal{S}}$):

$$\begin{aligned} player(X,Y,Z) &\leftarrow s_1(X,Y,Z,W) \\ team(X,Y,Z) &\leftarrow s_2(X,Y,Z) \\ team(X,Y,Z) &\leftarrow s_3(X,Y,Z) \\ coach(X,Y,Z) &\leftarrow s_4(X,Y,Z) \end{aligned}$$

□

The formal semantics of a data integration system $\mathcal{I}$ is defined with respect to a given instance $\mathcal{D}$ of the source schema, $\mathcal{S}$, in terms of the set $sem(\mathcal{I}, D)$ of all instances $\mathcal{B}$ of the global schema, $\mathcal{G}$, which satisfy all constraints in $\mathcal{G}$ and, moreover,

- $\{\boldsymbol{c} \mid \mathcal{B} \models q_{\mathcal{G}}(\boldsymbol{c})\} \subseteq \{\boldsymbol{c} \mid \mathcal{D} \models q_{\mathcal{S}}(\boldsymbol{c})\}$ for each mapping assertion of form (1), and
- $\{\boldsymbol{c} \mid \mathcal{D} \models q_{\mathcal{S}}(\boldsymbol{c})\} \subseteq \{\boldsymbol{c} \mid \mathcal{B} \models q_{\mathcal{G}}(\boldsymbol{c})\}$ for each mapping assertion of form (2),

where for any constants $\boldsymbol{c}$ on *Dom*, $\mathcal{DB} \models q(\boldsymbol{c})$ denotes that $q(\boldsymbol{c})$ evaluates to true on the database $\mathcal{DB}$.

The notions of *sound*, *complete*, and *exact mapping* between query expressions $q_{\mathcal{G}}(\boldsymbol{x})$ and $q_{\mathcal{S}}(\boldsymbol{x})$ [17], reflecting assumptions on the source contents, are then elegantly captured as follows:

- sound mapping: $q_{\mathcal{S}}(\boldsymbol{x}) \sqsubseteq q_{\mathcal{G}}(\boldsymbol{x})$ (intuitively, some data in the sources is missing),
- complete mapping: $q_{\mathcal{G}}(\boldsymbol{x}) \sqsubseteq q_{\mathcal{S}}(\boldsymbol{x})$ (intuitively, the sources contain excess data),
- exact mapping: $q_{\mathcal{S}}(\boldsymbol{x}) \sqsubseteq q_{\mathcal{G}}(\boldsymbol{x}) \wedge q_{\mathcal{G}}(\boldsymbol{x}) \sqsubseteq q_{\mathcal{S}}(\boldsymbol{x})$

The answer to a query Q with out query predicate $q(\boldsymbol{x})$ against a data integration system $\mathcal{I}$ with respect to source data $\mathcal{D}$, is given by the set of tuples $ans(Q, \mathcal{I}, \mathcal{D}) = \{\boldsymbol{c} \mid \mathcal{B} \models q(\boldsymbol{c})$, for each $\mathcal{B} \in sem(\mathcal{I}, \mathcal{D})\}$; that is, $ans(Q, \mathcal{I}, \mathcal{D})$ collects all tuples $\boldsymbol{c}$ on *Dom* such that $q(\boldsymbol{c})$ is a skeptical consequence with respect to all "legal" global databases.

In a GAV setting under sound mappings, the smallest candidate database $\mathcal{B}$ for $sem(\mathcal{I}, \mathcal{D})$, is given by the *retrieved global database*, $ret(\mathcal{I}, \mathcal{D})$, which is the materialization of all the views on the sources. Under exact mappings, $ret(\mathcal{I}, \mathcal{D})$ is in fact the only candidate database for $sem(\mathcal{I}, \mathcal{D})$.

Example 3. (cont'd) Note that the mapping $\mathcal{M}$ in our running example is a sound mapping. The global database $\mathcal{B}_0 = ret(\mathcal{I}, \mathcal{D})$ for $\mathcal{D}$ as in Figure 1 is shown in Figure 2. It violates the key constraint on $team$, witnessed by the two facts $team(\mathit{RM}, \mathit{Roma}, 10)$ and $team(\mathit{RM}, \mathit{Real\ Madrid}, 10)$, which coincide on $Tcode$ but differ on $Tname$. Since this key constraint is violated in every database $\mathcal{B}'$ which contains $\mathcal{B}_0$, it follows that $sem(\mathcal{I}, \mathcal{D})$ is empty, i.e., the global relations can not be consistently populated. □

global database $\mathcal{B}_0 = ret(\mathcal{I}, \mathcal{D})$

player:

10	*Totti*	*RM*
9	*Beckham*	*MU*

team:

RM	*Roma*	10
MU	*ManU*	8
RM	*Real Madrid*	10

coach:

7	*Camacho*	*RM*

source database $\mathcal{D}$

s_1:

10	*Totti*	*RM*	27
9	*Beckham*	*MU*	28

s_2:

RM	*Roma*	10

s_3:

MU	*ManU*	8
RM	*Real Madrid*	10

s_4:

7	*Camacho*	*RM*

Fig. 2. Global database $\mathcal{B}_0 = ret(\mathcal{I}, \mathcal{D})$ for the soccer scenario as retrieved from the sources

repair $\mathcal{R}_1$

player:

10	*Totti*	*RM*
9	*Beckham*	*MU*

team:

RM	*Roma*	10
MU	*ManU*	8

coach:

7	*Camacho*	*RM*

repair $\mathcal{R}_2$

player:

10	*Totti*	*RM*
9	*Beckham*	*MU*

team:

MU	*ManU*	8
RM	*Real Madrid*	10

coach:

7	*Camacho*	*RM*

Fig. 3. Repairs in the example data integration scenario $\mathcal{I}$ w.r.t. $\mathcal{D}$

Since "ex-falso-quodlibet" is not a desirable principle for database query answers (e.g., in our scenario (*Roma*) would be a query answer), relaxations aim at adopting global databases $\mathcal{B}$ which (1) satisfy all constraints and (2) satisfy the mapping assertion $\mathcal{M}$ with respect to $\mathcal{D}$ *as much as possible*. The latter may be defined in terms of a preference ordering (i.e., a reflexive and transitive relation) $\succeq$ over global databases $\mathcal{B}$, such that those $\mathcal{B}$ are accepted which are most preferred. Different possibilities for instantiating $\succeq$ exist and have been considered in a number of papers [7,8,19,10,11,12,20,13,21,15]. A popular one with GAV mappings is the one which prefers databases $\mathcal{B}$ which are as close as possible, under symmetric set difference, to the retrieved global databases $ret(\mathcal{I}, \mathcal{D})$; it amounts to the smallest set of tuples to be added and/or deleted. Other orderings are based on giving preferences to retaining tuples in $ret(\mathcal{I}, \mathcal{D})$ over adding new ones (as in loosely sound semantics [20,13]), or even demand that only deletion of tuples is allowed (guided by some completeness assumption [21]). These databases are commonly referred to as "repairs" of the global database $ret(\mathcal{I}, \mathcal{D})$.

Example 4. (cont'd) Let us adopt the preference relation $\succeq_{\mathcal{B}_0}$ which prefers $\mathcal{B}_1$ over $\mathcal{B}_2$ if the symmetric difference $\mathcal{B}_1 \triangle \mathcal{B}_0$ is a subset of $\mathcal{B}_2 \triangle \mathcal{B}_0$. Then the retrieved global database $\mathcal{B}_0$ for the running example has two repairs $\mathcal{R}_1$ and $\mathcal{R}_2$ shown in Figure 3.

Accordingly, the example query Q evaluates to $ans(Q, \mathcal{I}, \mathcal{D}) = \{(8), (9), (10)\}$, since the respective facts $q(\boldsymbol{c})$ are derived over both repairs. □

For a more rigorous discussion of the above model, in particular from the perspective of logic, we refer to [18], where it is also shown that a number of different orderings $\succeq$ used by different authors can be generically captured.

3 Employing Non-monotonic Logic Programs

Recently, several approaches to formalize repair semantics by using non-monotonic logic programs have been proposed, cf. [7,8,9,19,10,11,12,13,14,15]. The idea common to most of these works is to encode the constraints Σ of the global schema $\mathcal{G}$ into a function-free logic program, Π, using unstratified negation and/or disjunction, such that the answer sets of this program [22] yield the repairs of the global database. Answering a user query, Q, then amounts to cautious reasoning over the logic program Π augmented with the query, cast into rules, and the retrieved database $\mathcal{B}$. For an elaborated discussion of these proposals, we refer to [23]. In [19], the authors consider data integration via abductive logic programming, where, roughly speaking, the repairs of a database are computed as abductive explanations.

In the following, we discuss some key features of answer set programs which can be fruitfully exploited for doing data integration via ASP.

High expressiveness. ASP is a host for complex database queries. ASP with disjunction has Π_2^P-expressiveness, which means that each database query with complexity in Π_2^P can be expressed in this formalism [24]. Such expressiveness is strictly required for query answering in some settings [13,21,25]; for example, under loosely-exact semantics in the presence of inclusion dependencies for fixed queries, but also for ad hoc queries under absence of such dependencies. And, ASP with disjunction has co-NEXP$^{\mathrm{NP}}$ program complexity [24], and thus very complex problems can be polynomially reduced to it (as for query answering, however, not always in a data-independent manner). For more on complexity and expressiveness of ASP, see [26,27].

Declarative language. ASP supports a fully declarative, rule-based approach to information integration. While languages such as Prolog also support declarative integration, the algorithm-oriented semantics makes it more difficult to design and understand integrations policies for the non-expert.

Nondeterminism. ASP engines have been originally geared towards model generation (i.e., computation of one, multiple, or all answer sets) rather than towards theorem proving. This is particularly useful for solving certain AI problems including model-based diagnosis, where given some observations and a background theory, a model is sought which reconciles the actual and predicted observations in terms of assumptions about faulty components applying Occam's Razor.

Repair semantics for query answering from inconsistent data bases and integration systems [7,8,10,11,12,13,15] is closely related to this diagnostic problem. Using an ASP engine, a particular repair for a corrupted database might be computed and installed in its place.

Proximity to database query languages. ASP programs are close to logic-based database query languages such as conjunctive queries, union of conjunctive queries, (plain) SQL, and datalog. This in particular facilitates a seamless integration of various components of a data integration system – query evaluation, database "repair", and database mapping – into a uniform language.

Executable specifications. Given the seamless integration of various components of a data integration system, an ASP program for information integration tasks can be regarded as an executable specification, which can be run on supplied input data. This has been elaborated in [14,23] where abstract logic specification for querying a GAV data integration $\mathcal{I}$ with a query Q has been considered in terms of a hierarchically composed disjunctive datalog program $\Pi_{\mathcal{I}}(Q) = \Pi_{\mathcal{M}} \cup \Pi_{\Sigma} \cup \Pi_{Q}$ such that (in simplified notation):

1. $ret(\mathcal{I}, \mathcal{D}) \rightleftharpoons \mathrm{AS}(\Pi_{\mathcal{M}} \cup \mathcal{D})$, where $\Pi_{\mathcal{M}}$ is a stratified normal datalog program, computing the mapping $\mathcal{M}$;
2. $rep_{\mathcal{I}}(\mathcal{D}) \rightleftharpoons \mathrm{AS}(\Pi_{\Sigma} \cup ret(\mathcal{I}, \mathcal{D}))$, where Π_{Σ} is an (unstratified resp. disjunctive) program computing the repairs, and
3. $ans(Q, \mathcal{I}, \mathcal{D}) = \{\boldsymbol{c} \mid q(\boldsymbol{c}) \in M$ for each $M \in \mathrm{AS}((\Pi_{\mathcal{M}} \cup \Pi_{\Sigma} \cup \Pi_{Q} \cup \mathcal{D})\}$, where Π_{Q} is a non-recursive safe datalog program with negation (defining the query output predicate q);

here, $\mathrm{AS}(\mathcal{P})$ are the answer sets of a program $\mathcal{P}$ and $\rightleftharpoons$ denotes a polynomial-time computable correspondence between two sets.

Language constructs. The original ASP language [22] has been enriched with a number of constructs, including different forms of constraints such as DLV's weak constraints, Smodels's choice rules and weight constraints, rule preferences (see e.g. [28] for a survey), and more recently aggregates (see [29,30,31] and references therein).

Aggregates, for instance, are very desirable for querying databases (SQL provides many features in this respect). Weak and weight constraints are convenient for specifying certain integration semantics, such as repairs based on cardinality (Hamming) distance of changes to the retrieved global database. Rule preferences might be exploited to expressing preferences in repair, and e.g. help to single out a canonical repair. Therefore, refinements and variants of standards proposals for integration semantics can be accomodated well.

Knowledge representation capability. ASP provides, thanks to the availability of facts and rules, different kinds of negation (strong and weak i.e. default negation), a rich language for representing knowledge. This makes ASP attractive for crafting special, domain dependent integration policies, in which intuitively a knowledge-base is run for determining the best integration result.

4 Emerging ASP Research Issues

While ASP is attractive as a logic specification formalism for data integration, and available ASP engines can be readily applied for rapid prototyping of experimental systems

that work well on small examples, it becomes quickly apparent that ASP technology needs to be seriously improved in order to meet the requirements of this application and make its usage feasible in practice. Among others, the following important issues emerge:

Scalability. In real data integration scenarios, one needs to deal with massive amounts of data (in the Gigabytes and beyond), rather than with a few tuples. A graceful scaling of an integration system's response time with respect to the amount of data processed is desired. A problem in this respect is that current ASP technology builds on program grounding, i.e., the reduction of non-ground programs to ground (propositional) programs which are then evaluated with special algorithms. Even though the grounding strategies of DLV and Smodels' grounder Lparse are highly sophisticated and avoid as much as possible the generation of "unnecessary rules," the grounding of a repair program over a large dataset will be ways too large to render efficient query answering. Therefore, optimization methods are needed which allow for handling large amounts of data. A call for this has been made e.g. in [11].

Nonground queries. Another issue is that as seen in the example scenario, queries to an integration system typically contain variables, all whose instances should be computed. ASP solvers, however, have been conceived for model computation rather than for query answering. Query answering, e.g. as originally supported in DLV, was limited to ground (variable-free) queries (which can be reduced to model computation resp. checking program consistency by simple transformations). The straightforward method of reducing a non-ground query by instantiation to a series of (separate) ground queries is not efficient, since roughly the system response time will be $O(\#gq * srt)$, where $\#gq$ is the number of ground queries and srt is the response time for a single ground query. Therefore, efficient methods for answering non-ground queries are needed.

Software interoperability. In data integration settings in practice, data is stored in multiple repositories, and often in heterogeneous formats. In a relational setting, such repositories will be managed by a commercial DBMS such as Oracle, DB2, SQLServer etc. Data exchange between a DBMS and an ASP engine via files or other operating systems facilities, as was the only possibility with early ASP systems, requires extra development effort and, moreover, is an obvious performance bottleneck. To facilitate efficient system interoperability, suitable interfaces from ASP solvers to DBMS must be provided, such as an ODBC interface as available in other languages. Furthermore, interfaces to other software for carrying out specific tasks in data integration (e.g., data cleaning, data presentation) are desirable.

These issues are non-trivial and require substantial foundational and software development work. Scalability and non-ground query answering are particularly challenging issues, which are not bound to the data integration application. Advances on them will be beneficial to a wide range of other applications as well.

The above issues have been driving some of the research on advancing and enhancing ASP technology in the last years, and in particular of the ASP groups at the University of Calabria and at TU Vienna. A number of results have been achieved, which are briefly summarized as follows.

- As for scalability and optimization, the magic set method (introduced in [32]) has been extended to disjunctive programs and tuned for data integration [33,34,35]; focusing techniques and optimization methods genuine to data integration are presented in [14,23]; different variants of repair programs have been examined for their suitability, in this context, recent notions and results on program equivalence [36,37,38,39] turned out to be a useful tool.
- Non-ground query answering is supported in the current releases of DLV.
- Interfacing of relational DBMS is supported by an ODBC interface in DLV [40], and a tight coupling between ASP engines and relational DBMSs has been conceived [41].

These results have been obtained in the course of the INFOMIX project, which is briefly presented in the next subsection, since to our knowledge it is the most comprehensive initiative to employ ASP in a data integration system.

4.1 The INFOMIX Project

INFOMIX [42] has been launched jointly by the ASP groups of the University of Calabria and TU Vienna, the data integration group at the University of Rome "La Sapienza," and Rodan Systems S.A., a Polish database software house, with the objective to provide powerful information integration for handling inconsistent and incomplete information, using computational logic tools as an implementation host for advanced reasoning tasks. The usage of an ASP engine like DLV or Smodels which is capable of handling non-ground programs and provides the expressiveness needed, appeared to be well-suited. However, the research issues mentioned above had to be addressed in order to make employment of ASP in realistic integration scenarios feasible beyond toy examples.

The INFOMIX prototype [43,44] is built on solid theoretical foundations, and implements the GAV approach under sound semantics. It offers the user a powerful query language which, as a byproduct of the usage of ASP, allows in certain cases also queries beyond recursion-free positive queries (which are those expressed by non-recursive ASP programs without negation), in particular stratified queries or queries with aggregates, depending on the setting of the integrity constraints of the global schema; the underlying complexity and undecidability frontier has been charted in [20]. Furthermore, INFOMIX provides the user with tools for specifying and managing a data integration scenario, as well as with a rich layer for accessing and transforming data from sources (possibly dispersed on the Internet) in heterogeneous formats (including relational format, XML, and HTML under constraints) into a homogenous format (conceptually, into a fragment of XML Schema) by data wappers.

At the INFOMIX core are repair logic programs for handling data inconsistencies, which are dynamically compiled by rewriting algorithms. Pruning and optimization methods are applied which aim at reducing the portion of data which need to be accessed for query answering. In particular, the usage of the ASP engine is constrained to the inconsistent data part which needs repair. Details about this can be found in paper and reports [42,13,14,35].

Compared to data integration systems with similar semantics, of which the most prominent are the Hippo [45] and ConQuer [46], INFOMIX is capable of handling a

much larger range of queries and constraint settings which are realistic in practice. This is exemplified by the INFOMIX Demo Scenario, in which data from various legacy databases and web pages of the University of Rome “La Sapienza” are integrated into a global view which has 14 relations and about 30 integrity constraints, including key constraints, inclusion and exclusion dependencies. Most of the 9 typical user queries in the Demo Scenario can’t be handled by Hippo; ConQuer can only handle key constraints, and thus is not applicable to the scenario. On the other hand, Hippo and ConQuer are very efficient and faster than INFOMIX on the specific settings which they can handle.

Thanks to improved ASP technology and the optimization techniques, INFOMIX is able to handle the queries in the Demo Scenario reasonably efficient within a few seconds for core integration time, and tends to scale gracefully. Without these improvements and optimizations, the performance is much worse and the system response time barely acceptable.

5 Discussion and Conclusion

As argued above, data integration can be seen as an interesting model application of Answer Set Programming. The results which have been obtained so far are encouraging, and show clear benefits of using ASP. We remark that an experimental comparison of computing database repairs with different logic-based methods – QBF, CLP, SAT, and ASP solvers – in a propositional setting is reported in [9], which shows that ASP performs very well. We suspect that in the realm of a relational setting (in which QBF and SAT solvers can’t be directly applied and require preliminary grounding), it behaves even more advantageous.

In spite of the advances that have been achieved on the issues in Section 4, research on them is by no means closed, and in fact a lot of more work is necessary.

Optimization of ASP programs is still at a rather early stage, and there is room for improvement. Currently, optimization is done at the level of ASP solvers, which employ internal optimization strategies that to some extent build on heuristics. Optimization at the “external” level, independent of a concrete ASP solver, is widely unexplored. Recent results on program equivalences (cf. [36,37,38,47,39] and references therein) might provide a useful basis for optimization methods. However, as follows from classic results in database theory, basic static optimization tasks for ASP programs are undecidable in very plain settings (cf. [48]), and thus a detailed study and exploration of decidable cases is needed.

Efficient non-ground query answering is an issue which is perhaps tied to a more fundamental issue concerning the architecture of current state-of-the-art answer set engines: it is unclear whether their grounding approach is well-suited as a computational strategy. Indeed, for programs in which predicate arities are bounded by a constant, non-ground query answering can be carried out in polynomial space (as follows from results in [49]) while current answer set engines use exponential space for such queries in general.

As for software interoperability, ASP engines need to interface a large range of other data formats besides relational data, including popular formats like XML and, more recently, also RDF. For XML data, ASP extensions are desired which allow to manipulate them conveniently. The Elog language [50] may be a guiding example in

this direction. In turn, integration of ASP solvers into more complex software systems needs also better support.

As for future developments of ASP and data integration, one interesting issue would be to realize an operational data integration system which is deployed in a concrete application. The results of INFOMIX are encouraging in this direction. Here, a hybrid system combining complementary approaches like those of Hippo, ConQuer, and INFOMIX would be an intriguing idea. As for the perspective of advanced data integration at the level of a full-fledged commercial DBMS, we feel that research is still at an early stage and industry seems not to be ready for immediate takeup.

There are several interesting directions for further research on the usage of ASP in data integration. Among them is powerful mediated data integration, as discussed e.g. in [19], and peer-to-peer data integration in a network of information systems [51]. Furthermore, applications of ASP in advanced integration of information sources which contain information beyond factual knowledge, and in data model and ontology management might be worthwhile to explore. Thanks to its rich knowledge representation capabilities, ASP might prove to be a valuable tool for developing declarative formalisms in these areas.

Acknowledgments. I am very grateful to Wolfgang Faber for comments on a draft of this paper. I would like to thank all the many colleagues with whom I had the pleasure to discuss about and work on issues on data integration using non-monotonic logic programming, and here in particular the INFOMIX project team. The views expressed here are, however, personal and not necessarily in line with those of INFOMIX folks. Furthermore, I am very grateful to the support of the European Commission of this work under contracts IST-2001-33570 INFOMIX, FET-2001-37004 WASP, and IST-2001-33123 CologNeT, and the Austrian Science Fund (FWF) project P18019-N04.

References

1. Hayes, H., Mattos, N.: Information on demand. DB2 Magazine **8** (2003)
2. Halevy, A.Y.: Data integration: A status report. In: Proc. 10. Conference on Database Systems for Business, Technology and Web (BTW 2003), LNI 26, GI (2003) 24–29
3. Bertossi, L., Chomicki, J.: Query answering in inconsistent databases. In Chomicki, J., van der Meyden, R., Saake, G., eds.: Logics for Emerging Applications of Databases. Springer (2003) 43–83
4. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV System for Knowledge Representation and Reasoning. ACM Transactions on Computational Logic, to appear. Available via `http://www.arxiv.org/ps/cs.AI/0211004`.
5. Simons, P., Niemelä, I., Soininen, T.: Extending and Implementing the Stable Model Semantics. Artificial Intelligence **138** (2002) 181–234
6. Lierler, Y., Maratea, M.: Cmodels-2: Sat-based answer set solver enhanced to non-tight programs. In: Proc. 7th Int'l Conf. on Logic Programming and Nonmonotonic Reasoning (LPNMR-7). LNCS 2923, Springer (2004) 346–350
7. Arenas, M., Bertossi, L.E., Chomicki, J.: Specifying and querying database repairs using logic programs with exceptions. In: Proc. 4th Int. Conf. on Flexible Query Answering Systems (FQAS 2000), Springer (2000) 27–41
8. Arenas, M., Bertossi, L.E., Chomicki, J.: Answer sets for consistent query answering in inconsistent databases. Theory and Practice of Logic Programming **3** (2003) 393–424

9. Arieli, O., Denecker, M., Nuffelen, B.V., Bruynooghe, M.: Database repair by signed formulae. In: Proc. 3rd Int'l Symp. on Foundations of Information and Knowledge Systems (FoIKS 2004). LNCS 2942, Springer (2004) 14–30
10. Bertossi, L.E., Chomicki, J., Cortes, A., Gutierrez, C.: Consistent answers from integrated data sources. In: Proc. 6th Int'l Conf. on Flexible Query Answering Systems (FQAS 2002). (2002) 71–85
11. Bravo, L., Bertossi, L.: Logic programming for consistently querying data integration systems. In: Proc. 18th Int'l Joint Conf. on Artificial Intelligence (IJCAI 2003). (2003) 10–15
12. Bravo, L., Bertossi, L.: Deductive databases for computing certain and consistent answers to queries from mediated data integration systems. Journal of Applied Logic **3** (2005) 329–367
13. Calì, A., Lembo, D., Rosati, R.: Query rewriting and answering under constraints in data integration systems. In: Proc. IJCAI 2003. (2003) 16–21
14. Eiter, T., Fink, M., Greco, G., Lembo, D.: Efficient evaluation of logic programs for querying data integration systems. In: Proc. 19th Int'l Conf. on Logic Programming (ICLP 2003). LNCS 2916, Springer (2003) 163–177
15. Greco, G., Greco, S., Zumpano, E.: A logic programming approach to the integration, repairing and querying of inconsistent databases. In: Proc. ICLP 2001. (2001) 348–364
16. Mattos, N.M.: Integrating information for on demand computing. In: Proc. 29th Int'l Conf. on Very Large Data Bases (VLDB 2003). (2003) 8–14
17. Lenzerini, M.: Data integration: A theoretical perspective. In: Proc. 21st ACM Symposium on Principles of Database Systems (PODS 2002). (2002) 233–246
18. Andrea Calí, D.L., Rosati, R.: A comprehensive semantic framework for data integration systems. Journal of Applied Logic **3** (2005) 308–328
19. Arieli, O., Denecker, M., Nuffelen, B.V., Bruynooghe, M.: Coherent integration of databases by abductive logic programming. J. Artificial Intelligence Research **21** (2004) 245–286
20. Calì, A., Lembo, D., Rosati, R.: On the decidability and complexity of query answering over inconsistent and incomplete databases. In: Proc. PODS 2003. (2003) 260–271
21. Chomicki, J., Marcinkowski, J.: Minimal-change integrity maintenance using tuple deletions. Information and Computation **197** (2005) 90–121
22. Gelfond, M., Lifschitz, V.: Classical negation in logic programs and disjunctive databases. New Generation Computing **9** (1991) 365–385
23. Eiter, T., Fink, M., Greco, G., Lembo, D.: Optimization methods for logic-based consistent query answering over data integration systems. Tech. Report INFSYS RR-1843-05-05, Institute of Information Systems, TU Vienna, Austria (2005). Extends [14]
24. Eiter, T., Gottlob, G., Mannila, H.: Disjunctive Datalog. ACM Trans. on Database Systems **22** (1997) 364–417
25. Greco, G., Greco, S., Zumpano, E.: A logical framework for querying and repairing inconsistent databases. IEEE Trans. Knowl. Data Eng. **15** (2003) 1389–1408
26. Dantsin, E., Eiter, T., Gottlob, G., Voronkov, A.: Complexity and Expressive Power of Logic Programming. ACM Computing Surveys **33** (2001) 374–425
27. Marek, V.W., Remmel, J.B.: On the expressibility of stable logic programming. Journal of the Theory and Practice of Logic Programming **3** (2003) 551–567
28. Delgrande, J.P., Wand, K., Schaub, T., Tompits, H.: A classification and survey of preference handling approaches in nonmonotonic reasoning. Computational Intelligence **20** (2004) 308–334
29. Gelfond, M.: Representing Knowledge in A-Prolog. In Kakas, A.C., Sadri, F., eds.: Computational Logic. Logic Programming and Beyond. LNCS 2408, Springer (2002) 413–451
30. Pelov, N.: Semantics of Logic Programs with Aggregates. PhD thesis, Katholieke Universiteit Leuven, Leuven, Belgium (2004)

31. Faber, W., Leone, N., Pfeifer, G.: Recursive Aggregates in Disjunctive Logic Programs: Semantics and Complexity. In: Proc. 9th European Conference on Logics in Artificial Intelligence (JELIA 2004). LNCS 3229, Springer (2004) 200–212
32. Bancilhon, F., Maier, D., Sagiv, Y., Ullman, J.D.: Magic Sets and Other Strange Ways to Implement Logic Programs. In: Proc. PODS 1986. (1986) 1–16
33. Greco, S.: Binding Propagation Techniques for the Optimization of Bound Disjunctive Queries. IEEE Trans. Knowledge and Data Engineering **15** (2003) 368–385
34. Cumbo, C., Faber, W., Greco, G.: Enhancing the magic-set method for disjunctive datalog programs. In: Proc. ICLP 2004 (2004) 371–385
35. Faber, W., Greco, G., Leone, N.: Magic sets and their application to data integration. In: Proc. 10th Int'l Conf. on Database Theory (ICDT 2005). LNCS 3363, Springer (2005) 306–320
36. Lifschitz, V., Pearce, D., Valverde, A.: Strongly equivalent logic programs. ACM Transactions on Computational Logic **2** (2001) 526–541
37. Lin, F.: Reducing strong equivalence of logic programs to entailment in classical propositional logic. In: Proc. 8th Int'l Conf. on Principles of Knowledge Representation and Reasoning (KR 2002), (2002) 170–176
38. Eiter, T., Fink, M., Tompits, H., Woltran, S.: Simplifying logic programs under uniform and strong equivalence. In: Proc. LPNMR-7. LNCS 2923, Springer (2004) 87–99
39. Eiter, T., Fink, M., Woltran, S.: Semantical characterizations and complexity of equivalences in answer set programming. ACM Transactions on Computational Logic, to appear. Tech. Rep. INFSYS RR-1843-05-01, TU Vienna, Austria (2005)
40. Calimeri, F., Citrigno, M., Cumbo, C., Faber, W., Leone, N., Perri, S., Pfeifer, G.: New DLV features for data integration. In: Proc. JELIA 2004. LNCS 3229, Springer (2004) 698–701
41. Leone, N., Lio, V., Terracina, G.: DLV DB: Adding efficient data management features to ASP. In: Proc. LPNMR-7. LNCS 2923, Springer (2004) 341–345
42. INFOMIX homepage (since 2001) `http://sv.mat.unical.it/infomix`.
43. Leone, N., et al.: The INFOMIX system for advanced integration of incomplete and inconsistent data. In: Proc. ACM SIGMOD 2005 Conference, ACM (2005) 915–917
44. Leone, N., et al.: Data integration: a challenging ASP application. In: Proc. LPNMR 2005. LNCS, Springer (2005). This volume
45. Chomicki, J., Marcinkowski, J., Staworko, S.: Computing consistent query answers using conflict hypergraphs. In: Proc. 13th ACM Conference on Information and Knowledge Management (CIKM-2004), ACM Press (2004) 417–426
46. Fuxman, A., Fazli, E., Miller, R.J.: ConQuer: Efficient management of inconsistent databases. In: Proc. ACM SIGMOD 2005 Conference, ACM (2005) 155-166
47. Eiter, T., Fink, M., Tompits, H., Woltran, S.: Strong and uniform equivalence in answer-set programming: Characterizations and complexity results for the non-ground case. In: Proc. 20th National Conference on Artificial Intelligence (AAAI '05) (2005)
48. Halevy, A.Y., Mumick, I.S., Sagiv, Y., Shmueli, O.: Static analysis in datalog extensions. Journal of the ACM **48** (2001) 971–1012
49. Eiter, T., Faber, W., Fink, M., Pfeifer, G., Woltran, S.: Complexity of model checking and bounded predicate arities for non-ground answer set programming. In: Proc. KR 2004 (2004) 377–387
50. Baumgartner, R., Flesca, S., Gottlob, G.: The Elog web extraction language. In: Proc. 8th Int'l Conf. on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR 2001). LNCS 2250, Springer (2001) 548–560
51. Calvanese, D., Giacomo, G.D., Lenzerini, M., Rosati, R.: Logical foundations of peer-to-peer data integration. In: Proc. PODS 2004 (2004) 241–251

Halo I: A Controlled Experiment for Large Scale Knowledge Base Development*

Jürgen Angele, Eddie Moench, Henrik Oppermann, and Dirk Wenke

Ontoprise GmbH, Amalienbadstr. 36, D-76228 Karlsruhe, Germany
{angele, moench, oppermann, wenke}@ontoprise.de

Abstract. Project Halo is a multi-staged effort, sponsored by Vulcan Inc, aimed at creating the Digital Aristotle (DA), an application that will encompass much of the world's scientific knowledge and be capable of applying sophisticated problem solving to answer novel questions. Vulcan envisions two primary roles for the Digital Aristotle: as a tutor to instruct students in the sciences, and as an interdisciplinary research assistant to help scientists in their work. As a first step towards this goal, there was a six-month Pilot phase, designed to assess the state of the art in applied Knowledge Representation and Reasoning (KR&R). Vulcan selected three teams, each of which was to formally represent 70 pages from the Advanced Placement (AP) chemistry syllabus and deliver knowledge based systems capable of answering questions on that syllabus. The evaluation quantified each system's *coverage* of the syllabus in terms of its ability to answer novel, previously unseen questions and to provide *human-readable* answer justifications. These justifications will play a critical role in building user trust in the question-answering capabilities of the Digital Aristotle.Despite differences in approach, all three systems did very well on the challenge, achieving performance comparable to the human median. The analysis also provided key insights into how the approaches might be scaled, while at the same time suggesting how the cost of producing such systems might be reduced.

1 Introduction

Today, the knowledge available to humankind is so extensive that it is not possible for a single person to assimilate it all. This is forcing us to become much more specialized, further narrowing our worldview and making interdisciplinary collaboration increasingly difficult. Thus, researchers in one narrow field may be completely unaware of relevant progress being made in other neighboring disciplines. Even within a single discipline, researchers often find themselves drowning in new results. MEDLINE®[1], for example, is an archive of 4,600 medical journals in thirty languages, containing over twelve million publications, with 2,000 added daily.

* Full support for this research was provided by Vulcan Inc. as part of Project Halo.

[1] MEDLINE is the National Library of Medicine's premier bibliographic database covering the fields of medicine, nursing, dentistry, veterinary medicine, the health care system, and the preclinical sciences.

C. Baral et al. (Eds.): LPNMR 2005, LNAI 3662, pp. 26 – 39, 2005.
© Springer-Verlag Berlin Heidelberg 2005

The final Digital Aristotle DA will differ from classical expert systems in four important ways:

- *Speed and ease of knowledge formulation:* Classical expert systems required years to perfect and highly skilled knowledge engineers to craft them; the DA will provide tools to facilitate rapid knowledge formulation by domain experts with little or no help from knowledge engineers.
- *Coverage:* Classical expert systems were narrowly focused on the single topic for which they were specifically designed; the DA will over time encompass much of the world's scientific knowledge.
- *Explanations:* Classical expert systems produced explanations derived directly from inference proof trees; the DA will produce concise explanations, appropriate to the domain and the user's level of expertise.

The Pilot phase of Project Halo was a 6-month effort to set the stage for a long-term research and development effort aimed at creating the Digital Aristotle. The primary objective was to evaluate the state of the art in applied KR&R systems. Understanding the performance characteristics of these technologies was considered to be especially critical to the DA, as they are expected to form the basis of its reasoning capabilities. The first objectives were to develop suitable evaluation methodologies; the project was also designed to help in the determination of a research and development roadmap for KR&R systems. Finally, the project adopted principles of scientific transparency aimed at producing understandable, reproducible results.

Three teams were contracted to participate in the evaluation: a team led by SRI International with substantial contributions from Boeing Phantom Works and the University of Texas at Austin; a team from Cycorp; and our team at ontoprise.

Significant attention was given to selecting a proper domain for the evaluation. In the end, a 70-page subset of introductory college-level Advanced Placement (AP) chemistry was selected because it was reasonably self-contained and did not require solutions to other hard AI problems, such as representing and reasoning with uncertainty, or understanding diagrams.

Fig. 1 lists the topics in the chemistry syllabus. Topics included: stoichiometry calculations with chemical formulas; aqueous reactions and solution stoichiometry; and chemical equilibrium. Background material was also identified to make the selected chapters more fully self-contained[2].

Subject	*Chapters*	*Sections*	*Pages*
Stoichiometry: Calculations with Chemical Formulas	3	3.1 – 3.2	75 - 83
Aqueous Reactions and Solution Stoichiometry	4	4.1 – 4.4	113 - 133
Chemical Equilibrium	16	16.1 – 16.11	613 - 653

Fig. 1. Course Outline for the Halo Challenge

[2] Sections 2.6-2.9 in chapter two provide detailed information. Chapter 16 also requires the definition of moles, which appears in section 3.4 pp 87-89, and molarity, which can be found on page 134. The form of the equilibrium expression can be found on page 580, and buffer solutions can be found in section 17.2.

This scope was large enough to support a large variety of novel, and hence unanticipated, question types. One analysis of the syllabus identified nearly 100 distinct chemistry laws, suggesting that it was rich enough to require complex inference. It was also small enough to be represented relatively quickly – which was essential because the three Halo teams were allocated only four months to create formal encodings of the chemistry syllabus. This amount of time was deemed sufficient to construct detailed solutions that leveraged the existing technologies, yet was too brief to allow significant revisions to the teams' platforms. Hence, by design, we were able to avoid undue customization to the task domain and thus to create a true evaluation of the state-of-the-art of KR&R technologies.

2 The Technology

The three teams had to address the same set of issues: knowledge formation, question answering, and explanation generation, [3], [4], [5]. They all built knowledge bases in a formal language and relied on knowledge engineers to encode the requisite knowledge. Furthermore, all the teams used automated deductive inference to answer questions.

2.1 Knowledge Formation

For testing purposes in the dry run Vulcan Inc. provided 50 syllabus questions for each team. The encoding of the corpus (70 pages from (Brown, LeMay and Bursten (2003))) has been done in the ontoprise team in three different phases. During the first phase the knowledge within the corpus has been encoded into the ontology and into rules, not regarding the 50 syllabus questions. This knowledge has been tested with some 40 exercises from (Brown, LeMay and Bursten (2003)). For each of these exercises an encoded input file containing the question in their formal language representation, and an output file containing the expected answer have been created. These pairs have been used for regression tests of the knowledge base.

In the next phase the syllabus questions have been tested with a covering of around 30% of these questions. During this phase the knowledge base has been refined until coverage of around 70% of these questions has been reached. Additionally, in parallel during this phase, the explanation rules have been encoded. In the so-called dry run itself, the encoded syllabus questions have been sent to Vulcan to test the installed systems. The remaining time to the challenge run has been used to refine the encoding of the knowledge base and the explanation rules. During the entire process, the library of test cases has been extended and used for automatic testing purposes. This ensured stability of the knowledge base against changes.

Cycorp used a hybrid approach by first concentrating on representing the basic concepts and principles of the corpus, and gradually shifting over to a question-driven approach. SRI's approach for knowledge formation was highly question driven. Starting from the 50 sample questions, they worked backwards to identify what pieces of knowledge will be needed to solve them.

2.2 The Architecture of the Knowledge Base

The resulting knowledge base has a three layered architecture. The upper level contains the ontological information about concepts like elements, reactions, substances and mixtures together with their attributes and relationships. This layer provides the domain vocabulary and the domain structure and is target of the queries posed to the system. This small ontology layer consisted of around 40 concepts, and 60 attributes and relations. For the second layer we identified around 35 basic chemical operations which are represented in around 400 F-Logic rules [19]. The third layer contains the basic facts like elements and the like represented in instances.

Let us illustrate this by an example. The ontology defines the concept mixture with attributes *hasPHValue*, *hasHConcentration, hasComponent*, *hasMole*, and *hasQuantity*. These attributes describe the formulae of components, the moles of the components and the quantities of the components. The attribute *hasPHValue* represents the ph-value of the mixture. Given such a mixture the following query queries for the H-concentration of the mixture:

```
FORALL X <- m1:Mixture[hasHConcentration->>X].
```

For the different types of mixtures different rules compute this H-concentration. E.g. for buffer solutions the H-concentration is described by the following rule which represents one basic chemical operation

```
rule3: FORALL BF,MB,SF,MS,H,B,Kb,S,KbMB,KbMBminus,MSKb,OH
bufferPh(BF,MB,SF,MS,H)
<- B:WeakBase[hasFormula->>BF; hasKb->>Kb] and ….
```

This basic chemical operation is now attached to the ontology and defines the attribute *hasHConcentration* of a mixture (there is an additional rule to compute the pH-value from this H-concentration):

```
rule2: FORALL M,H,F1,F2,M1,M2
M:BufferSolution[hasHConcentration->>H] <-
M:Mixture[hasComponents->>{F1,F2}; hasMoles-
>>{c(F1,M1),c(F2,M2)}] and bufferPh(F1,M1,F2,M2,H).
```

The advantages of this three level architecture are twofold. First the basic chemical operations are nearly independent from each other and can therefore be developed and tested independently. So each chemical operation is a stand-alone knowledge chunk.

Accessing the values using an ontology on the other hand frees the encoder from knowing all the specialized predicates and enables an access to the different information pieces in a way that is much closer to natural language than the predicates of the basic chemical operations.

2.3 The Inferencing Process

Figure 2 shows question 44 from the Multiple Choice section of the exam along with the answer generated by the ontoprise system.

Question 44
Original question

Given is an aqueous solution containing equimolar ratios of the following pairs of substances.

a. CH3COOH and NaCH3COO
b. HClO4 and NaCl
c. KOH and HCl
d. NH3 and KOH
e. CH3NH2 and CH3NH3Cl

Which is a buffer with a pH > 7 ?

Answer: E

Fig. 2. Question 44 from the Multiple Choice section of the Challenge Exam, and the answer generated by our system

This question of the challenge run focuses on acid-base reactions (ch. 4.3 Brown, LeMay and Bursten (2003)) and on buffer solutions (ch. 17.2 Brown, LeMay and Bursten (2003)). In the following we present those parts of the knowledge base which contributed to the correct answer E and the chemical knowledge behind them.

First of all this question has been encoded in F-Logic. For each possible choice, the encoding looks like the following example for E:

```
m:Mixture[hasComponents->>{"CH3NH2"," CH3NH3Cl"}; has-
Moles->>{c("CH3NH2",1),c("CH3NH3Cl",1)}].
```

```
FORALL M answer("E") <- M:Mixture[hasPHValue->>Ph] and
M:BufferSolution and greater(Ph,7).
```

The first two lines encode the premise, viz. that a mixture *m* exists consisting of two substances with 1 mole each. The third and fourth lines constitute a query asking for the pH-value of the buffer and whether this pH-value is greater than 7.

There is a rule in the KB which computes the Ph-value of a mixture from the H-concentration using the well-known equation *Ph = -log H (eq.4)*:

```
rule1: FORALL M,Ph,H M[hasPHValue->>Ph]
<- M:Mixture[hasHConcentration->>H] and phH(H,Ph).
```

The H-concentration is computed by rule3:

```
rule3: FORALL
BF,MB,SF,MS,H,B,Kb,S,KbMB,KbMBminus,MSKb,OH
bufferPh(BF,MB,SF,MS,H) <-
B:WeakBase[hasFormula->>BF; hasKb->>Kb] and
S:IonicMolecule[hasFormula->>SF] and
isSaltOf(SF,BF) and multiply(Kb,MB,KbMB) and multi-
ply(KbMB,-1,KbMBminus) and add(MS,Kb,MSKb) and quad-
ratic(OH,1,MSKb,KbMBminus) and multiply(H,OH,1.0E-14).
```

Consider this rule in more detail. The first three lines of the body of the rule determine whether BF is the formula of a weak base and whether SF is a salt of this weak base BF and thus checks whether we have a buffer solution consisting of a weak base and its salt. The second part of the rule body then determines first the OH-concentration and then uses the relation *H * OH = 1.0E-14 (eq.3)* to determine the H-concentration. The OH-concentration is given by the Kb-value of the weak acid and by the moles of the base MB and the salt MS (cf. ch. 17.2 Brown, LeMay and Bursten (2003), eq. 1,2):

*OH = -(Kb+MS) + sqrt((Kb+MS)2+4*Kb*MB))/2*

This formula computes the OH-concentration in our case to *4.4 E-4 (eq.2)*. This results in an H-concentration of 2.27E-11 according to the formula *H = 1.0E-14/OH (eq.3)*. Finally the pH-value is thus determined to 10.64 according to *Ph = -log H (eq.4).* Rule 3 gets its input *BF=CH3NH2, MB=1, SF=CH3NH3Cl, MS=1* top-down propagated over the head, determining whether we have a buffer solution consisting of a weak base and its salt and then computes the H-concentration. This H-concentration is then propagated bottom-up to rule2 which assigns the H-concentration to the mixture. Finally rule2 computes the pH-value from the H-concentration and assigns this pH-value to the mixture and thus delivers the first part of the query.

This proof tree together with the intermediate instantiations of the variables is shown in figure 3.

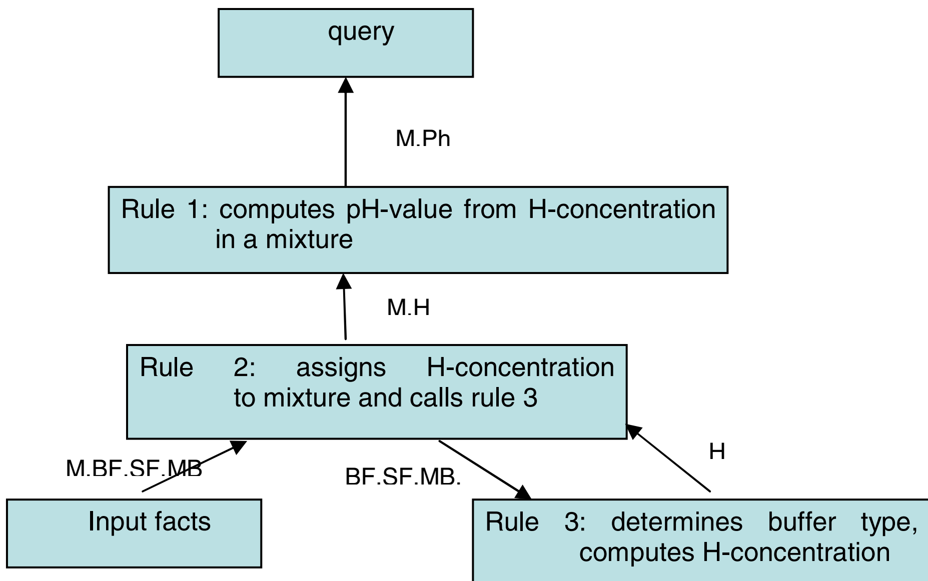


Fig. 3. The proof tree shows the rule dependencies in our example case

An analysis of the proof trees for the syllabus questions showed that the depth of these proof trees ranged from 1 up to 20. These proof trees are stored in a infrencing protocol (as F-Logic facts) to be used for generating explanations.

Our inference system OntoBroker™ provides means for efficient reasoning with instances and for the capability to express arbitrary powerful rules, e.g. ones that quantify over the set of classes. As F-Logic relies on the well-founded semantics [17] a possible implementation is the alternating fixed point procedure. This is a forward chaining method which computes the entire model for the set of rules, i.e. the set of true and unknown facts. For answering a query according to its implementation, the entire model must be computed (if possible) to derive the variable substitutions for the query. In contrast, our inference engine OntoBroker™ performs a mixture of forward and backward chaining based on the dynamic filtering algorithm [18] to compute (the smallest possible) subset of the model for answering the query. In most cases this is much more efficient than simple evaluation strategies. These techniques stem from the deductive data base community and are optimized to deliver all answers instead of one single answer as e.g. resolution does.

2.4 The Answer Justification

The mentioned inferencing process generated for question 44 the following justification:

If 1 moles of the **weak base CH3NH2** are mixed with 1 moles of **its salt CH3NH3Cl**, the concentration of OH−atoms in the **resulting buffer solution** is 4.4E−4 and thus the **ph−value is 10.64**

> The [H+] concentration multiplied with the [OH−] concentration is always the **ion−product constant of water 1.0E−14**. In this case the values are **[H+]=2.27E−11 and [OH−]=4.4E−4**.
>
> > The equation for calculating the ph−value is **ph=−log[H+]**. Thus we get ph−value **ph = 10.64, H+ concentration [H+] = 2.27E−11**.

The explanations are generated in a second inference run by explanation rules working on the inference protocol (F-Logic facts). To each of the important rules in the knowledge base such an explanation rule is assigned:

```
FORALL I,EX1,HP,OHM explain(EX1,I) <-
I:Instantiation[ofRule->>phvaluephOH; instantiatedVars-
>>{i(H,HP),i(OH,OHM)}] and
EX1 is ("The [H+] concentration multiplied with the
[OH-] concentration is always the <b>ion-product con-
stant of water 1.0E-14</b>. In this case the values are
<b>[H+]="+HP+" and [OH-]="+OHM+"</b>.").
```

The first two lines of the rule body access the fact generated by the first inference run, i.e. the variables of the original rule and its instantiations. The second part generates the textual explanation. These explanation rules resemble the explanation templates of the SRI system.

Our system formatted the answers and the justifications into a pdf document which was the input for grading by the chemistry professors after the challenge run. Figure 4 shows the concrete output for question 20.

Question 20

Original question

The pH of a 1.0M solution of HCl is:

a. 1.0
b. 0.1
c. 0.0
d. less than zero
e. between 0 and 1

Answer

C

Justification

- The equation for calculating the ph–value is **ph=–log[H+]**. Thus we get ph–value **ph = 0, H+ concentration [H+] = 1.**
- In the given mixture **the concentration of H3O–molecules(atoms) is 1 M.**
  - ◊ **HCl is a strong monoprotic acid** that completely splits into H+ and Cl–. A 1–molar mixture of HCl has therefore a 1–molar concentration of H+ and Cl– molecules.

This is a logical, well ordered presentation.

Fig. 4. Output for multiple choice question 20. Note the output's format: the question number is indicated at the top; followed by the full English text of the original question; next, the letter answer is indicated; finally, the answer justification is presented. The graders written remarks are included.

3 Evaluation

3.1 The Experiment

At the end of four months, knowledge formulation was stopped, even though the teams had not completed the task. All three systems were sequestered on identical servers at Vulcan. Then the challenge exam, consisting of 100 novel AP-style English questions, was released to the teams. The exam consisted of three sections: 50 multiple-choice questions and two sets of 25 multipart questions—the detailed answer and free form sections. The detailed answer section consisted mainly of quantitative questions requiring a "fill in the blank" (with explanation) or short essay response. The

free form section consisted of qualitative, comprehension questions, which exercised additional reasoning tasks such as meta-reasoning, and relied more, if only in a limited way, on commonsense knowledge and reasoning.

Due to the limited scope of the Pilot, there was no requirement that questions be input in their original, natural language form. Thus, two weeks were allocated to the teams for the translation of the exam questions into their respective formal languages. Upon completion of the encoding effort, the formal question encodings of each team were evaluated by a program-wide committee to guarantee high fidelity to the original English.

Once the encodings were evaluated, Vulcan personnel submitted them to the sequestered systems. The evaluations ran in batch mode. Each of the three systems produced an output file in accordance with a pre-defined specification. Vulcan engaged three chemistry professors to evaluate the exams. Adopting an AP-style evaluation methodology, they graded each question for both correctness and the quality of the explanation. The exam encompassed 168 distinct gradable components consisting of questions and question sub-parts.

3.2 Empirical Results

In general, all teams showed up with similar good scores in the challenge run. There was no time limit for the answering process, so that the teams' systems did not take an AP exam under original conditions. The aim of the run was to test the general ability of the systems to answer questions of a complex domain.

3.3 Performance

Processing performance of OntoNova (the ontoprise system[4]) has been much faster than its competitors, depending on the systems used to perform the challenge run. Our system for the official challenge run required about an hour to process the encodings. For a second, optional and not sequestrated run we slightly improved the knowledge base and brought down processing time to less than 10 minutes.

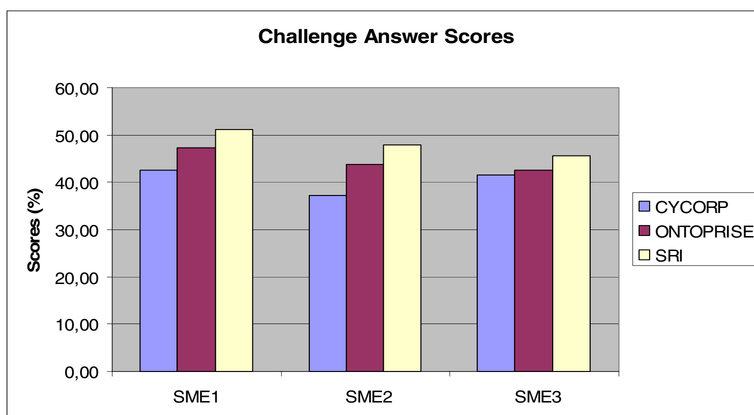


Fig. 5. Challenge answer scores

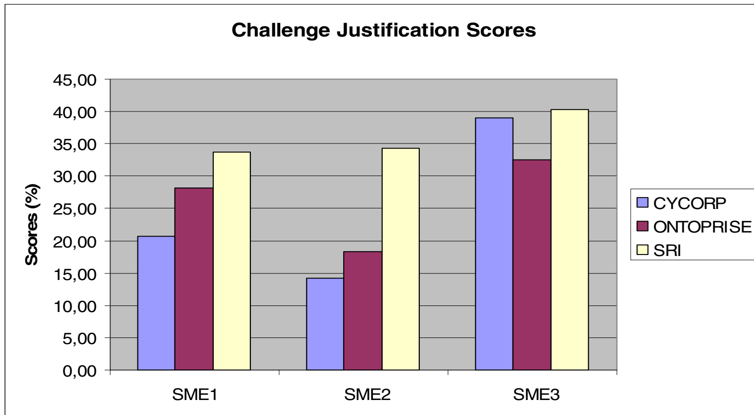


Fig. 6. Challenge justification scores

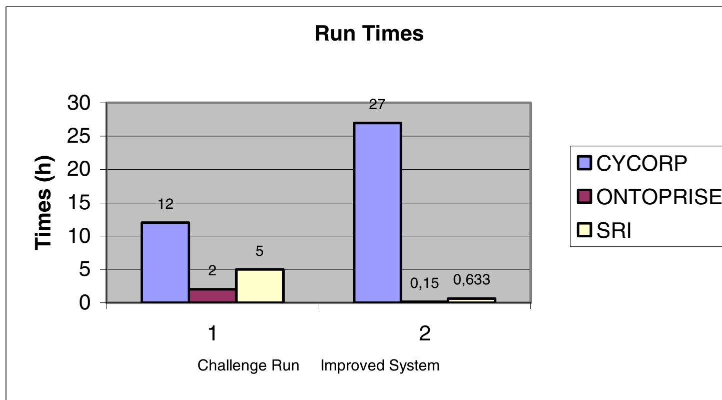


Fig. 7. Run times

4 Lessons Learned

The analysis of failures showed that ontoprise lost points for mainly the following reasons:

1. The knowledge base contained a lot of typos, like wrongly or undefined variables in rules, faulty connections from explanation rules to their corresponding rules, lacking explanation rules etc., which prevented rules and explanation rules from firing.
2. The knowledge base contains faults of the contents itself, like faulty assignments of substances to categories.
3. The knowledge base did not entirely cover the domain.
4. The knowledge base contained the knowledge in principle but this knowledge could not be exploited.

5. Explanations have been sometimes too lengthy and redundant.
6. The knowledge base did not contain enough "common sense" knowledge to answer unexpected or very general queries. This especially holds for the free form section.
7. Some of the questions where out of scope in the opinion of ontoprise.

The first point will in future be tackled by improving the modeling environment OntoStudio™ and by introspection. OntoStudio™ will in future allow parallel formation of axioms and their justification and will support tracking changes from rules into explanation rules and vice versa. This will solve many cases of justification brittleness. The reasoning capabilities of OntoBroker™ cannot "only" be used for inferencing proper and answer justification, but also for maintaining consistency in the ontology and the knowledge base. In particular, we have investigated the Halo I knowledge base and found that reification of inference rules together with some simple checks ("Does every justification rule have a core inference rule?", "Is every constant declared to be a constant and not a misspelled variable?" etc.) would have discovered simple errors. The removal of these simple errors would have improved correctness of answers by about 20% – to a record of around 50% correct answers overall!

The points 2 and 3 will in future be supported by an enhanced testing environment. We used test cases consisting of queries to the knowledge base and expected answers to keep the knowledge base stable after modifications. This testing will be integrated into OntoStudio™ making it easier for the knowledge engineer to collect test cases and to run them against the knowledge base. Additionally it became clear that the knowledge base must be tested against a lot more examples which could not be done in Halo I due to the time restriction.

Basically rules in F-Logic allow a lot of flexibility in different ways to query them. One major problem for failures of point 4 has been the built-ins used. Ontobroker™ contains around 150 builtins and a lot of these built-ins are exploitable in a restricted way only. For instance the builtin for solving a quadratic equation could only be used for computing the resulting x-values, given the values a,b,c. Computing the value a, given x,b, and c was not incorporated. These built-ins will be reengineered to make them as flexible as possible.

Point 5 will be tackled by advanced answer justification. Additional steps will be: (i) integration of additional knowledge, (ii) reduction of redundancies of explanations, and (iii) abstraction from fine-grained explanations. In detail:

(i), when the proof tree does not contain enough information in itself, additional information will be needed to create comprehensible explanations. In the context of chemistry, e.g., the proof tree contains the formulae of the different substances. However, in the explanations the names of the formulae are often necessary to produce understandable justifications. The generation of names from formulae then requires additional information not found within the proof tree.

(ii), in knowledge bases different paths often lead to the same result. In chemistry, e.g., the acidity order of two substances may be determined by the pH-values that may be computed from the amount of substances in aqueous solutions – or they may be given by qualitative criteria like the number of oxygen elements in the formulae of the substances. Both criteria lead to the same ordering. As a consequence there are two different explanation paths for the same result. An explanation rule may rank these two explanation paths and ignore one of them when formulating the justification.

(iii), often it is necessary to come to different abstractions of explanations. For example for the purpose of debugging the knowledge base a very fine grained explanation level is necessary while for the grader a lower grained explanation level is more reasonable. The single inference steps and thus the proof tree which is a representation of these single inference steps provide the finest resolution of explanations. Additional rules may summarize these steps and may thus produce higher abstracted explanations. E.g. in the chemistry domain such rules may summarize the three different steps of a reaction like the (a) decomposition of the substances into their ionic compounds, (b) the composition of an insoluble substance out of these compounds and (c) the precipitation of the resulting substance from the solution as a precipitation reaction.

Point 6 can to our opinion only be tackled by modeling more "common sense" knowledge. Still it has to be proven whether this is really true. Cycorp says that it incorporates a lot of such "common sense" knowledge in his knowledge base, but they have not been much better in the free form section compared to ontoprise. At least it is clear that the inference engine must be able to handle a larger knowledge base. The performance results have shown that ontoprise has an inference engine with a very high performance compared to the other teams.

5 Next Steps

As we noted at the beginning of this article, Project halo is a multi-staged effort. In the foregoing, we have described Phase I, which assessed the capability of knowledge-based systems to answer a wide variety of unanticipated questions with coherent explanations. Phase II of Project Halo will examine whether tools can be built to enable domain experts to build such knowledge based systems with an ever-decreasing reliance on knowledge engineers. In 2004, ontoprise entered Phase II of Project Halo with a multi-national team, bringing together best-of-breed cutting edge technology and expertise: the DARKMATTER team (Deep Authoring, Answering and Representation of Knowledge by Subject Matter Experts).

The DARKMATTER approach to the Halo-2 challenge can be described as a Semantic Web in the Small. Tim Berners-Lee, the inventor of the World Wide Web, envisions the Semantic Web as a global cooperation of individuals contributing data, ontology and logical rules to support sophisticated machine-generated answers from the Web. The DARKMATTER team adopts relevant Semantic Web principles to be applied in the small, while abstracting from some of the riskier and more intricate issues associated with the Semantic Web (like dynamic trust). The Semantic Web in the small is document-rooted, i.e. a lot of knowledge is acquired from existing content by reformulating it in formal terms. The DARKMATTER approach improves upon existing technology, allowing subject matter experts to interact with formal scientific knowledge by themselves in an intuitive way. The goal of DARKMATTER STUDIO, the system we are proposing, is to allow knowledge formation by subject matter experts (SMEs) at a cost of less than $1000 per page of input text — comparable to the costs estimated by publishers for producing a page of textbook text. The DARKMATTER QUERYINTERFACE has been designed to support questioning of the system by users with no training in knowledge formalization. Together, the DARKMATTER components will provide a system

able to answer AP-like exam questions, achieving at least an AP level 3 score — the sophistication achieved by Halo-1 systems.

To reach these objectives, DARKMATTER STUDIO and DARKMATTER QUERYINTERFACE are designed along four dimensions. First, our analysis has shown what knowledge types occur in AP exams in biology, chemistry and physics. Second, SMEs have different cognitive abilities they may draw upon when working with knowledge, for example interacting with texts, diagrams or visualizations of conceptual structures. Correspondingly, we provide user interfaces that support these different modalities. Third, formulating knowledge and utilizing it constitutes a knowledge life cycle with a number of steps, viz. knowledge entry, justification authoring, testing, debugging, validation, querying and presentation. Configurations of user interfaces (so called Perspectives) work together to support the major different steps in the knowledge formation and questioning life cycle. Fourth, Semantic Web technologies provide the representation and reasoning platform in which knowledge formation is restricted to instantiating and refining ontologies and linking between knowledge modules aka Semantic Web pages in the small.

Each member of the DARKMATTER team has been selected in order to reflect the challenging nature of the Halo-2 undertaking: Ontoprise for system integration, knowledge representation and reasoning as well as knowledge acquisition, Carnegie-Mellon University (CMU) for natural language processing; DFKI (German institute for AI) for intelligent user interfaces and usability; Georgia Institute of Technology for understanding and reasoning with diagrams; and University of Brighton, UK, for intuitive querying.

Project Halo in general, and DARKMATTER in particular, may become for the field of knowledge representation what the World Wide Web became for hypertext. Just as the WWW protocols built in an clever way on top of existing infrastructure, DARKMATTER builds on existing frameworks with over 30 years of research to make the theoretical achievements of knowledge representation and reasoning available for a wide audience in a Semantic Web in the Small.

Just as the WWW was of immediate value to its users, even at the time when only one single Web server existed, it will be crucial to the success of DARKMATTER that it provides immediate added value to its users. To do so, DARKMATTER builds on a comprehensive analysis investigating the capabilities and possible needs of its future user base. Halo-2 starts at an appropriate time. At present, the Semantic Web at large targets a representational infrastructure with objectives that coincide with those of project Halo. It must not be forgotten, however, that the value of the original WorldWideWeb was not created just by the defined protocols, but also by tools that were easy to use, like Mosaic. In this sense, DARKMATTER represents the knowledge formulation and question answering system of a future scientific SemanticWeb. Being an interface used to teach and interact 15 with the Digital Aristotle, DARKMATTER may become the equivalent of Front Page and Internet Explorer for the Semantic Web.

Just as the WWW has changed the way we retrieve texts and work online, a system like DARKMATTER has the potential to change the way that scientists and engineers do research or teaching in the future.

References

1. Voorhees, E.M., *The TREC-8 Question Answering Track Report*, in *The Eighth Text REtrieval Conference (TREC-8)*. 1999. p. 77-82.
2. Brown, T.L., *et al.*, *Chemistry: The Central Science*. 2003, New Jersey: Prentice Hall.
3. Barker, K., *et al. A Question-answering System for AP Chemistry: Assessing KRR Technologies*. in *9th International Conference on Principles of Knowledge Representation and Reasoning*. 2004. Whistler, Canada.
4. Angele, J., *et al.*, *Ontology-based Query and Answering in Chemistry: Ontonova @ Project Halo*, . 2003.
5. Witbrock, M. and G. Matthews, *Cycorp Project Halo Final Report.* 2003.
6. Barker, K., B. Porter, and P. Clark, *A Library of Generic Concepts for Composing Knowledge Bases*, in *Proc. 1st Int Conf on Knowledge Capture (K-Cap'01)*. 2001. p. 14--21.
7. Novak, G., *Conversion of Units of Measurement.* IEEE Transactions on Software Engineering, 1995. **21**(8): p. 651-661.
8. Clancey, W.J., *The epistemology of a rule-based expert system: A framework for explanation.* Artificial Intelligence, 1983. **20**: p. 215-251.
9. Voorheese, E.M., *Overview of TREC Question Answering Task*, . 2001.
10. Chaudhri, V. and R. Fikes, eds. *AAAI Fall Symposium on Question Answering Systems*. . 1999, AAAI.
11. Neches, R., *et al.*, *Enabling Technology for Knowledge Sharing.* AI Magazine, 1991. **12**(3): p. 36--56.
12. Cohen, P., *et al.*, *The DARPA High Performance Knowledge Bases Project.* AI Magazine, 1998. **19**(4): p. 25--49.
13. Brachman, R.J., *et al.*, *Reducing CLASSIC to ``Practice": Knowledge Representation Theory Meets Reality.* Artificial Intelligence Journal, 1999. **114**: p. 203-237.
14. Keyes, J., *Why Expert Systems Fail?* IEEE Expert, 1989. **4**: p. 50-53.
15. Batanov, D. and P. Brezillon, eds. *First International Conference on Successes and Failures of Knowledge-based Systems in Real World Applications*. . 1996, Asian Institute of Technology: Bangkok, Thailand.
16. Pool, M., J.F. K. Murray, M. Mehrotra, R. Schrag, J. Blythe,, and H.C. J. Kim, P. Miraglia, T. Russ, and D. Schneider. *Evaluation of Expert Knowledge Elicited for Critiquing Military Courses of Action.* in *Proceedings of the Second International Conference on Knowledge Capture (KCAP-03)*. 2003.
17. A. Van Gelder, K. A. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. Journal of the ACM, 38(3):620–650, July 1991.
18. M. Kifer and E. Lozinskii. A framework for an efficient implementation of deductive databases. In Proceedings of the 6th Advanced Database Symposium, pages 109–116, Tokyo, August 1986.
19. M. Kifer, G. Lausen, and J.Wu. Logical foundations of object-oriented and framebased languages. Journal of the ACM, 42:741–843, 1995.

Unfounded Sets for Disjunctive Logic Programs with Arbitrary Aggregates*

Wolfgang Faber

Department of Mathematics, University of Calabria, 87030 Rende (CS), Italy
faber@mat.unical.it

Abstract. Aggregates in answer set programming (ASP) have recently been studied quite intensively. The main focus of previous work has been on defining suitable semantics for programs with arbitrary, potentially recursive aggregates. By now, these efforts appear to have converged. On another line of research, the relation between unfounded sets and (aggregate-free) answer sets has lately been rediscovered. It turned out that most of the currently available answer set solvers rely on this or closely related results (e.g., loop formulas).

In this paper, we unite these lines and give a new definition of unfounded sets for disjunctive logic programs with arbitrary, possibly recursive aggregates. While being syntactically somewhat different, we can show that this definition properly generalizes all main notions of unfounded sets that have previously been defined for fragments of the language.

We demonstrate that, as for restricted languages, answer sets can be crisply characterized by unfounded sets: They are precisely the unfounded-free models. This result can be seen as a confirmation of the robustness of the definition of answer sets for arbitrary aggregates. We also provide a comprehensive complexity analysis for unfounded sets, and study its impact on answer set computation.

1 Introduction

The introduction of aggregate atoms [1,2,3,4,5,6,7,8] is one of the major linguistic extensions to Answer Set Programming of the recent years. While both semantic and computational properties of standard (aggregate-free) logic programs have been deeply investigated, relatively few works have focused on logic programs with aggregates; some of their semantic properties and their computational features are still far from being fully clarified.

The proposal for answer set semantics in [8] seems to be receiving a consensus. Recent works, such as [9,10] give further support for the plausibility of this semantics by relating it to established constructs for aggregate-free programs. In particular, [9] presented a semantics for very general programs, and showed that it coincides with both answer sets of [8] and Smodels answer sets (the latter holds for weight constraints with positive weights only). In [10] the notion of unfounded sets is extended from aggregate-free programs to programs with aggregates in a conservative way, retaining important

* This work was supported by an APART grant of the Austrian Academy of Sciences and the European Commission under projects IST-2002-33570 INFOMIX, IST-2001-37004 WASP.

C. Baral et al. (Eds.): LPNMR 2005, LNAI 3662, pp. 40–52, 2005.
© Springer-Verlag Berlin Heidelberg 2005

semantical and computational properties. It should be noted that unfounded sets are the basis of virtually all currently available ASP solvers [11,4,12,13,14,15]. Extending this notion to programs with aggregates should therefore be seen as paving the way to effective and efficient systems for programs with aggregates.

However, in [10] only a fragment of the language has been considered, namely nondisjunctive programs with monotone and antimonotone aggregates. In this paper we lift this restriction and define unfounded sets for *disjunctive* programs with *arbitrary* aggregates. To this end, some substantial change in the definition is necessary to account for nonmonotone aggregates. Nevertheless, we are able to prove that our definition is a clean extension of all main previous notions of unfounded sets: On the respective fragments, our unfounded sets always coincide with the previously proposed ones.

Importantly, we can show that our notion of unfounded sets crisply characterizes both models and answer sets of [8] for arbitrary programs. We also study complexity issues for unfounded sets and put them into perspective with respect to complexity of reasoning tasks on answer sets for various fragments of programs with aggregates. Finally, we discuss the impact of our results on computation.

Summarizing, our contributions are as follows:

- We define the notion of unfounded sets for *disjunctive* logic programs with *arbitrary* aggregates. We demonstrate that this notion is a sound generalization of all main previous concepts of unfounded sets.
- We analyze the properties of our unfounded sets, which parallel those of previous definitions. We show that a unique greatest unfounded set always exists for the class of unfounded-free interpretations.
- We characterize answer sets in terms of unfounded sets. One of the results is that a model is an answer sets iff it is unfounded-free.
- We study the complexity of determining unfounded-freeness of an interpretation, and deduce the complexity for answer set checking, which turns out to be a crucial factor for the complexity of query answering.
- We indicate applications of our results; in particular, they allow to conceive how to build efficient systems for computing answer sets for programs with aggregates.

2 Logic Programs with Aggregates

2.1 Syntax

We assume that the reader is familiar with standard LP; we refer to the respective constructs as *standard atoms, standard literals, standard rules*, and *standard programs*. Two literals are said to be complementary if they are of the form p and `not` p for some atom p. Given a literal L, $\neg.L$ denotes its complementary literal. Accordingly, given a set A of literals, $\neg.A$ denotes the set $\{\neg.L \mid L \in A\}$. For further background, see [16,17].

Set Terms. A DLP$^{\mathcal{A}}$*set term* is either a symbolic set or a ground set. A *symbolic set* is a pair $\{\,Vars : Conj\}$, where $Vars$ is a list of variables and $Conj$ is a conjunction of

standard atoms.[1] A *ground set* is a set of pairs of the form $\langle \overline{t} : Conj \rangle$, where $\overline{t}$ is a list of constants and $Conj$ is a ground (variable free) conjunction of standard atoms.

Aggregate Functions. An *aggregate function* is of the form $f(S)$, where S is a set term, and f is an *aggregate function symbol*. Intuitively, an aggregate function can be thought of as a (possibly partial) function mapping multisets of constants to a constant.

Example 1. In the examples, we adopt the syntax of DLV *to denote aggregates.* Aggregate functions currently supported by the DLV system are: #count (number of terms), #sum (sum of non-negative integers), #times (product of positive integers), #min (minimum term), #max (maximum term)[2].

Aggregate Literals. An *aggregate atom* is $f(S) \prec T$, where $f(S)$ is an aggregate function, $\prec \in \{=, <, \leq, >, \geq\}$ is a predefined comparison operator, and T is a term (variable or constant) referred to as guard.

Example 2. The following aggregate atoms are in DLV notation, where the latter contains a ground set and could be a ground instance of the former:

$$\#\texttt{max}\{Z : r(Z), a(Z, V)\} > Y \qquad \#\texttt{max}\{\langle 2 : r(2), a(2, k)\rangle, \langle 2 : r(2), a(2, c)\rangle\} > 1$$

An *atom* is either a standard atom or an aggregate atom. A *literal* L is an atom A or an atom A preceded by the default negation symbol not; if A is an aggregate atom, L is an *aggregate literal*.

DLP$^{\mathcal{A}}$ **Programs.** A DLP$^{\mathcal{A}}$ *rule* r is a construct

$$a_1 \ \vee \ \cdots \ \vee \ a_n \ \text{:-} \ b_1, \ldots, b_k, \ \texttt{not} \ b_{k+1}, \ldots, \ \texttt{not} \ b_m.$$

where $a_1, \cdots, a_n$ are standard atoms, $b_1, \cdots, b_m$ are atoms, and $n \geq 1$, $m \geq k \geq 0$. The disjunction $a_1 \ \vee \ \cdots \ \vee \ a_n$ is referred to as the *head* of r while the conjunction $b_1, ..., b_k, \ \texttt{not} \ b_{k+1}, ..., \texttt{not} \ b_m$ is the *body* of r. We denote the set of head atoms by $H(r)$, and the set $\{b_1, ..., b_k, \ \texttt{not} \ b_{k+1}, ..., \texttt{not} \ b_m\}$ of the body literals by $B(r)$. $B^+(r)$ and $B^-(r)$ denote, respectively, the set of positive and negative literals in $B(r)$. Note that this syntax does not explicitly allow integrity constraints (rules without head atoms). They can, however, be simulated in the usual way by using a new symbol and negation.

A DLP$^{\mathcal{A}}$ *program* is a set of DLP$^{\mathcal{A}}$ rules. In the sequel, we will often drop DLP$^{\mathcal{A}}$, when it is clear from the context. A *global* variable of a rule r appears in a standard atom of r (possibly also in other atoms); all other variables are *local* variables.

Safety. A rule r is *safe* if the following conditions hold: (i) each global variable of r appears in a positive standard literal in the body of r; (ii) each local variable of r appearing in a symbolic set $\{Vars : Conj\}$ appears in an atom of $Conj$; (iii) each guard of an aggregate atom of r is a constant or a global variable. A program $\mathcal{P}$ is safe if all $r \in \mathcal{P}$ are safe. In the following we assume that DLP$^{\mathcal{A}}$ programs are safe.

[1] Intuitively, a symbolic set $\{X : a(X, Y), p(Y)\}$ stands for the set of X-values making $a(X, Y), p(Y)$ true, i.e., $\{X \mid \exists Y s.t.\ a(X, Y), p(Y) \ is\ true\}$.

[2] The first two aggregates roughly correspond, respectively, to the cardinality and weight constraint literals of Smodels. #min and #max are undefined for empty set.

2.2 Answer Set Semantics

Universe and Base. Given a DLP$^{\mathcal{A}}$ program $\mathcal{P}$, let $U_{\mathcal{P}}$ denote the set of constants appearing in $\mathcal{P}$, and $B_{\mathcal{P}}$ be the set of standard atoms constructible from the (standard) predicates of $\mathcal{P}$ with constants in $U_{\mathcal{P}}$. Given a set X, let $\overline{2}^X$ denote the set of all multisets over elements from X. Without loss of generality, we assume that aggregate functions map to $\mathbb{I}$ (the set of integers).

Example 3. $\#$count is defined over $\overline{2}^{U_{\mathcal{P}}}$, $\#$sum over $\overline{2}^{\mathbb{N}}$, $\#$times over $\overline{2}^{\mathbb{N}^+}$, $\#$min and $\#$max are defined over $\overline{2}^{\mathbb{N}} - \{\emptyset\}$.

Instantiation. A *substitution* is a mapping from a set of variables to $U_{\mathcal{P}}$. A substitution from the set of global variables of a rule r (to $U_{\mathcal{P}}$) is a *global substitution for r*; a substitution from the set of local variables of a symbolic set S (to $U_{\mathcal{P}}$) is a *local substitution for S*. Given a symbolic set without global variables $S = \{Vars : Conj\}$, the *instantiation of S* is the following ground set of pairs $inst(S)$:

$\{\langle\gamma(Vars) : \gamma(Conj)\rangle \mid \gamma$ *is a local substitution for S*$\}$.[3]

A *ground instance* of a rule r is obtained in two steps: (1) a global substitution σ for r is first applied over r; (2) every symbolic set S in $\sigma(r)$ is replaced by its instantiation $inst(S)$. The instantiation $Ground(\mathcal{P})$ of a program $\mathcal{P}$ is the set of all possible instances of the rules of $\mathcal{P}$.

Interpretations. An *interpretation* for a DLP$^{\mathcal{A}}$ program $\mathcal{P}$ is a consistent set of standard ground literals, that is $I \subseteq (B_{\mathcal{P}} \cup \neg.B_{\mathcal{P}})$ such that $I \cap \neg.I = \emptyset$. A standard ground literal L is true (resp. false) w.r.t I if $L \in I$ (resp. $L \in \neg.I$). If a standard ground literal is neither true nor false w.r.t I then it is undefined w.r.t I. We denote by I^+ (resp. I^-) the set of all atoms occurring in standard positive (resp. negative) literals in I. We denote by $\bar{I}$ the set of undefined atoms w.r.t. I (i.e. $B_{\mathcal{P}} \setminus I^+ \cup I^-$). An interpretation I is *total* if $\bar{I}$ is empty (i.e., $I^+ \cup \neg.I^- = B_{\mathcal{P}}$), otherwise I is *partial*.

An interpretation also provides a meaning for aggregate literals. Their truth value is first defined for total interpretations, and then generalized to partial ones.

Let I be a total interpretation. A standard ground conjunction is true (resp. false) w.r.t I if all its literals are true (resp. false). The meaning of a set, an aggregate function, and an aggregate atom under an interpretation, is a multiset, a value, and a truth-value, respectively. Let $f(S)$ be a an aggregate function. The valuation $I(S)$ of S w.r.t. I is the multiset of the first constant of the elements in S whose conjunction is true w.r.t. I. More precisely, let $I(S)$ denote the multiset $[t_1 \mid \langle t_1, ..., t_n : Conj\rangle \in S \wedge$ *Conj is true w.r.t. I*$]$. The valuation $I(f(S))$ of an aggregate function $f(S)$ w.r.t. I is the result of the application of f on $I(S)$. If the multiset $I(S)$ is not in the domain of f, $I(f(S)) = \bot$ (where $\bot$ is a fixed symbol not occurring in $\mathcal{P}$).

An instantiated aggregate atom A of the form $f(S) \prec k$ is *true w.r.t. I* if: (i) $I(f(S)) \neq \bot$, and, (ii) $I(f(S)) \prec k$ holds; otherwise, A is false. An instantiated aggregate literal not $A =$ not $f(S) \prec k$ is *true w.r.t. I* if (i) $I(f(S)) \neq \bot$, and, (ii) $I(f(S)) \prec k$ does not hold; otherwise, A is false.

[3] Given a substitution σ and a DLP$^{\mathcal{A}}$ object Obj (rule, set, etc.), we denote by $\sigma(Obj)$ the object obtained by replacing each variable X in Obj by $\sigma(X)$.

If I is a *partial* interpretation, an aggregate literal A is true (resp. false) w.r.t. I if it is true (resp. false) w.r.t. *each total* interpretation J extending I (i.e., $\forall$ J s.t. $I \subseteq J$, A is true (resp. false) w.r.t. J); otherwise it is undefined.

Example 4. Consider the atom $A = \#\mathtt{sum}\{\langle 1{:}p(2,1)\rangle, \langle 2{:}p(2,2)\rangle\} > 1$. Let S be the ground set in A. For the interpretation $I = \{p(2,2)\}$, each extending total interpretation contains either $p(2,1)$ or $\mathtt{not}\ p(2,1)$. Therefore, either $I(S) = [2]$ or $I(S) = [1,2]$ and the application of $\#\mathtt{sum}$ yields either $2 > 1$ or $3 > 1$, hence A is true w.r.t. I.

Remark 1. Our definitions of interpretation and truth values preserve "knowledge monotonicity". If an interpretation J extends I (i.e., $I \subseteq J$), then each literal which is true w.r.t. I is true w.r.t. J, and each literal which is false w.r.t. I is false w.r.t. J as well.

Minimal Models. Given an interpretation I, a rule r is *satisfied w.r.t. I* if some head atom is true w.r.t. I whenever all body literals are true w.r.t. I. A total interpretation M is a *model* of a DLP$^{\mathcal{A}}$ program $\mathcal{P}$ if all $r \in Ground(\mathcal{P})$ are satisfied w.r.t. M. A model M for $\mathcal{P}$ is (subset) minimal if no model N for $\mathcal{P}$ exists such that $N^+ \subset M^+$. Note that, under these definitions, the word *interpretation* refers to a possibly partial interpretation, while a *model* is always a total interpretation.

Answer Sets. We now recall the generalization of the Gelfond-Lifschitz transformation and answer sets for DLP$^{\mathcal{A}}$ programs from [8]: Given a ground DLP$^{\mathcal{A}}$ program $\mathcal{P}$ and a total interpretation I, let $\mathcal{P}^I$ denote the transformed program obtained from $\mathcal{P}$ by deleting all rules in which a body literal is false w.r.t. I. I is an answer set of a program $\mathcal{P}$ if it is a minimal model of $Ground(\mathcal{P})^I$.

Example 5. Consider interpretation $I_1 = \{p(a)\}$, $I_2 = \{\mathtt{not}\ p(a)\}$ and two programs $P_1 = \{p(a) :- \#\mathtt{count}\{X : p(X)\} > 0.\}$ and $P_2 = \{p(a) :- \#\mathtt{count}\{X : p(X)\} < 1.\}$.

$Ground(P_1) = \{p(a) :- \#\mathtt{count}\{\langle a : p(a)\rangle\} > 0.\}$ and $Ground(P_1)^{I_1} = Ground(P_1)$, $Ground(P_1)^{I_2} = \emptyset$. Furthermore, $Ground(P_2) = \{p(a) :- \#\mathtt{count}\{\langle a : p(a)\rangle\} < 1.\}$, and $Ground(P_2)^{I_1} = \emptyset$, $Ground(P_2)^{I_2} = Ground(P_2)$ hold.

I_2 is the only answer set of P_1 (since I_1 is not a minimal model of $Ground(P_1)^{I_1}$), while P_2 admits no answer set (I_1 is not a minimal model of $Ground(P_2)^{I_1}$, and I_2 is not a model of $Ground(P_2) = Ground(P_2)^{I_2}$).

Note that any answer set A of $\mathcal{P}$ is also a model of $\mathcal{P}$ because $Ground(\mathcal{P})^A \subseteq Ground(\mathcal{P})$, and rules in $Ground(\mathcal{P}) - Ground(\mathcal{P})^A$ are satisfied w.r.t. A.

Monotonicity. Given two interpretations I and J we say that $I \leq J$ if $I^+ \subseteq J^+$ and $J^- \subseteq I^-$. A ground literal ℓ is *monotone*, if for all interpretations I, J, such that $I \leq J$, we have that: (i) ℓ true w.r.t. I implies ℓ true w.r.t. J, and (ii) ℓ false w.r.t. J implies ℓ false w.r.t. I. A ground literal ℓ is *antimonotone*, if the opposite happens, that is, for all interpretations I, J, such that $I \leq J$, we have that: (i) ℓ true w.r.t. J implies ℓ true w.r.t. I, and (ii) ℓ false w.r.t. I implies ℓ false w.r.t. J. A ground literal ℓ is *nonmonotone*, if it is neither monotone nor antimonotone.

Note that positive standard literals are monotone, whereas negative standard literals are antimonotone. Aggregate literals may be monotone, antimonotone or nonmonotone, regardless whether they are positive or negative. Nonmonotone literals include the sum over (possibly negative) integers and the average.

3 Unfounded Sets

We now give a definition of unfounded set for arbitrary DLP$^{\mathcal{A}}$ programs. It should be noted that it is not possible to just take over the previous definitions in [18,11,10], as all of them make a distinction on the kind of atoms, be it positive and negative atoms, or the generalized version of monotone and antimonotone atoms. Just as in [8], where the same problem with the transformation of the program was lifted, we need to introduce a novel definition, which does not distinguish between the kinds of atoms.

In the following we denote by $S1 \dot{\cup} \neg.S2$ the set $(S_1 \setminus S_2) \cup \neg.S_2$, where S_1 and S_2 are sets of standard ground literals.

Definition 1 (Unfounded Set). *A set X of ground atoms is an unfounded set for a program $\mathcal{P}$ w.r.t. an interpretation I if, for each rule r in $Ground(\mathcal{P})$ having some atoms from X in the head, at least one of the following conditions holds:*

1. *some literal of $B(r)$ is false w.r.t. I,*
2. *some literal of $B(r)$ is false w.r.t. $I \dot{\cup} \neg.X$, or*
3. *some atom of $H(r) \setminus X$ is true w.r.t. I.*

Intuitively, conditions 1 and 3 state that rule satisfaction does not depend on the atoms in X, while condition 2 ensures that the rule is satisfied also if the atoms in X are switched to false. Note that $\emptyset$ is always an unfounded set, independent of interpretation and program.

Example 6. Let interpretation $I_0 = \emptyset$ and $\mathcal{P} = \{a(0) \vee a(1) \text{ :- } \#\texttt{avg}\{X : a(X)\} = 1.,$ $a(2) \vee a(1) \text{ :- } \#\texttt{avg}\{X : a(X)\} = 1.\}$. The unfounded sets w.r.t. I_0 are $\emptyset$, $\{a(0), a(1)\}$, $\{a(1), a(2)\}$, and $\{a(0), a(1), a(2)\}$. Only condition 2 applies in these cases. For $I_1 = \{a(0)\}$, $\{a(1), a(2)\}$, and $\{a(0), a(1), a(2)\}$ are unfounded sets. For $I_2 = \{a(1)\}$, $\{a(0)\}$, $\{a(2)\}$, $\{a(0), a(2)\}$, and $\{a(0), a(1), a(2)\}$ are unfounded sets. For $I_3 = \{\texttt{not}\ a(0), \texttt{not}\ a(1)\}$, $\{a(0), a(1), a(2)\}$ and all of its subsets are unfounded sets.

In the sequel, we will demonstrate the robustness of Def. 1, and show that some crucial properties of unfounded sets of nondisjunctive, aggregate-free programs continue to hold, while a few others do not, basically mirroring unfounded sets for disjunctive, aggregate-free programs. We first show that Def. 1 is a generalization of a previous definition of unfounded sets for aggregate-free programs:

Theorem 1. *For an aggregate-free program $\mathcal{P}$ and interpretation I, any unfounded set w.r.t. Def. 1 is an unfounded set as defined in [11].*

Proof. Recall that a set X of ground atoms is unfounded w.r.t. Def. 3.1 of [11], if at least one of the following conditions holds for each rule r in $Ground(P)$ having some atoms from X in the head: (a) $B(r) \cap \neg.I \neq \emptyset$, or (b) $B^+(r) \cap X \neq \emptyset$, or (c) $(H(r) \setminus X) \cap I \neq \emptyset$. On the other hand, in the aggregate-free case, Def. 1 amounts to: (1) $B(r) \cap \neg.I \neq \emptyset$, or (2) $B(r) \cap \neg.(I \dot{\cup} \neg.X) \neq \emptyset$, or (3) $(H(r) \setminus X) \cap I \neq \emptyset$.

Obviously, (a) is equivalent to (1), and (c) is equivalent to (2). Now observe that $B^+(r) \cap X \neq \emptyset$ implies $B(r) \cap X \neq \emptyset$, which implies $B(r) \cap (\neg.(I \setminus X) \cup X) \neq \emptyset$ which is equivalent to $B(r) \cap \neg.((I \setminus X) \cup \neg.X) \neq \emptyset \Leftrightarrow B(r) \cap \neg.(I \dot{\cup} \neg.X) \neq \emptyset$, so

(b) implies (2). On the other hand, (2) is equivalent to $B(r) \cap ((\neg.I \setminus \neg.X) \cup X) \neq \emptyset$, and therefore (A) $B(r) \cap (\neg.I \setminus \neg.X) \neq \emptyset$ or (B) $B(r) \cap X \neq \emptyset$ holds. (A) clearly implies (a), and (B) implies (b) because X contains only atoms, hence $B^-(r) \cap X = \emptyset$. In total, (2) implies (a) or (b). □

By Proposition 3.3 in [11], unfounded sets of [11] generalize the "original" unfounded sets of [18], which were defined for nondisjunctive programs. Therefore it follows from Theorem 1 that also unfounded sets of Def. 1 generalize those of [18] on nondisjunctive programs without aggregates.

Corollary 1. *For a nondisjunctive, aggregate-free program P and interpretation I, any unfounded set w.r.t. Def. 1 is a standard unfounded set (as defined in [18]).*

Recently, unfounded sets have been defined for nondisjunctive programs with monotone and antimonotone aggregates in [10]. Def. 1 also generalizes this notion.

Theorem 2. *For a nondisjunctive program P with only monotone and antimonotone aggregates and interpretation I, any unfounded set w.r.t. Def. 1 is an unfounded set w.r.t. [10].*

Proof. A set X of ground atoms is unfounded w.r.t. [10], if at least one of the following conditions holds for each rule r in $Ground(P)$ having some atoms from X in the head: (a) some antimonotone body literal of r is false w.r.t. I, and (b) some monotone body literal of r is false w.r.t. $I \dot{\cup} \neg.X$.

We first observe that condition 3 is always false for nondisjunctive programs, as $H(r) \setminus X = \emptyset$, since $H(r) \cap X \neq \emptyset$ and $|H(r)| \leq 1$.

Now, observe that $I \dot{\cup} \neg.X \leq I$ holds. So, if a monotone body literal of r is false w.r.t. I, it is also false w.r.t. $I \dot{\cup} \neg.X$, and if an antimonotone body literal of r is false w.r.t. $I \dot{\cup} \neg.X$, it must be false w.r.t. I. Therefore, if condition 1 holds for a monotone literal, also condition 2 and (b) hold for this literal; conversely, if condition 2 holds for an antimonotone literal, also condition 1 and (a) hold for it. So, since (a) and (b) trivially imply condition 1 and 2, respectively, we obtain equivalence. □

The union of two unfounded sets of nondisjunctive, aggregate-free programs is guaranteed to be an unfounded set as well. For disjunctive programs, this does not hold; also the addition of nonmonotone aggregates invalidates this property.

Observation 3. *If X_1 and X_2 are unfounded sets for a program $\mathcal{P}$ w.r.t. I, then $X_1 \cup X_2$ is not necessarily an unfounded set for $\mathcal{P}$ w.r.t. I, even if $\mathcal{P}$ is nondisjunctive.*

Example 7. Consider $I = \{a(0), a(1), a(-1)\}$ and $\mathcal{P} = \{a(1) :- \#\mathtt{avg}\{X : a(X)\} = 0.,$ $a(-1) :- \#\mathtt{avg}\{X : a(X)\} = 0., a(0).\}$. Both $\{a(1)\}$ and $\{a(-1)\}$ are unfounded sets for $\mathcal{P}$ w.r.t. I, while $\{a(1), a(-1)\}$ is not unfounded.

In this example, some elements in unfounded sets occur also in the interpretation. This is not a coincidence, as shown by the following proposition.

Proposition 1. *If X_1 and X_2 are unfounded sets for a program $\mathcal{P}$ w.r.t. I and both $X_1 \cap I = \emptyset$ and $X_2 \cap I = \emptyset$ hold, then $X_1 \cup X_2$ is an unfounded set for $\mathcal{P}$ w.r.t. I.*

Proof. Consider a rule r where $H(r) \cap X_1 \neq \emptyset$ (symmetric arguments hold for X_2). At least one of the conditions of Def. 1 holds w.r.t. X_1. We will show that the conditions also hold w.r.t. $X_1 \cup X_2$.

If condition 1 holds w.r.t. X_1, then it trivially holds also for $X_1 \cup X_2$. If condition 2 holds, a body literal is false w.r.t. $I \dot\cup \neg.X_1$, so it is false w.r.t. $I \cup \neg.X_1$ (since $I \cap X_1 = \emptyset$). Because of Remark 1, it is then also false w.r.t. $I \cup \neg.X_1 \cup \neg.X_2 = I \dot\cup \neg.(X_1 \cup X_2)$. If condition 3 holds, some atom a of $H(r) \setminus X_1$ is true w.r.t. I, so $a \in I$. It follows that $a \notin X_2$, and so $a \in H(r) \setminus (X_1 \cup X_2)$ is still true w.r.t. I. □

We next define interpretations which never contain any element of their unfounded sets.

Definition 2 (Unfounded-free Interpretation). *Let I be an interpretation for a program $\mathcal{P}$. I is unfounded-free if $I \cap X = \emptyset$ for each unfounded set X for $\mathcal{P}$ w.r.t. I.*

For unfounded-free interpretations, Prop. 1 holds for all unfounded sets.

Corollary 2. *If X_1 and X_2 are unfounded sets for a program $\mathcal{P}$ w.r.t. an unfounded-free interpretation I, then also $X_1 \cup X_2$ is an unfounded set for $\mathcal{P}$ w.r.t. I.*

We can therefore define the *Greatest Unfounded Set* (GUS) for unfounded-free interpretations as the union of all unfounded sets. Note that for non-unfounded-free interpretations, there is in general no unique GUS, as demonstrated in Ex. 7.

Definition 3. *Given a program $\mathcal{P}$ and an unfounded-free interpretation I, let $GUS_{\mathcal{P}}(I)$ (the GUS for $\mathcal{P}$ w.r.t. I) denote the union of all unfounded set for $\mathcal{P}$ w.r.t. I.*

These features are shared with disjunctive logic programs without aggregates, as discussed in [11]. However, while aggregate- and disjunction-free programs possess a unique GUS for arbitrary interpretations, Ex. 7 shows that this does not hold for disjunction-free programs with aggregates. By virtue of Thm. 2 and Thm. 10 of [10], which states that a unique GUS exists for nondisjunctive programs with monotone and antimonotone aggregates, we can infer that the presence of nonmonotone aggregates or disjunction and a non-unfounded-free interpretation is necessary to invalidate the existence of a unique GUS.

4 Unfounded Sets and Answer Sets

We will now use the notion of unfounded sets to characterize models and answer sets. We begin with models and show that the negative part of a model is an unfounded set and vice versa.

Theorem 4. *Given a total interpretation I and program $\mathcal{P}$, I^- is an unfounded set for $\mathcal{P}$ w.r.t. I iff I is a model of $\mathcal{P}$.*

Proof. $(\Rightarrow)$: For any rule, either (i) $H(r) \cap I^- = \emptyset$, or (ii) $H(r) \cap I^- \neq \emptyset$. If (i), then $H(r) \cap I \neq \emptyset$, i.e. the head is true and r is satisfied w.r.t. I. If (ii) then one of the conditions of Def. 1 must hold. If condition 1 holds, the body is false w.r.t. I and r is satisfied w.r.t. I. If condition 2 holds, a body literal is false w.r.t. $I \dot\cup \neg.I^- = I$, so it

coincides with condition 1. If condition 3 holds, $H(r) \cap I \neq \emptyset$, and therefore the rule is satisfied w.r.t. I. In total, if I^- is an unfounded set for $\mathcal{P}$ w.r.t. I, all rules are satisfied w.r.t. I, hence I is a model of $\mathcal{P}$.

$(\Leftarrow)$: If I is a model, all rules are satisfied, so for any rule r, either (i) $H(r) \cap I \neq \emptyset$ or (ii) if $H(r) \cap I = \emptyset$ then a body literal l is false w.r.t. I. So also for any rule r with $H(r) \cap I^- \neq \emptyset$, either (i) or (ii) holds. If (i), then condition 3 of Def. 1 applies. If (ii), then condition 1 (and also condition 2, since $I \mathbin{\dot\cup} \neg.I^- = I$) applies. Therefore I^- is an unfounded set. □

We now turn to answer sets. Each answer set is a model, so its negative part is an unfounded set. We can show that it is the greatest unfounded set. Conversely, if the negative part of a total interpretation is its greatest unfounded set, it is an answer set.

Theorem 5. *A total interpretation I is an answer set of $\mathcal{P}$ iff $I^- = GUS_{\mathcal{P}}(I)$*[4].

Proof. $(\Rightarrow)$: If I is an answer set, it is also a model of $\mathcal{P}$, so by Thm. 4, I^- is an unfounded set for $\mathcal{P}$ w.r.t. I. We next show that I is unfounded-free w.r.t. $\mathcal{P}$, from which $I^- = GUS_{\mathcal{P}}(I)$ follows. Let us assume an unfounded set X for $\mathcal{P}$ w.r.t. I exists such that $I \cap X \neq \emptyset$. We can show that then $I \mathbin{\dot\cup} \neg.X$ is a model of $\mathcal{P}^I$, contradicting the fact that I is an answer set of $\mathcal{P}$.

First note that for any rule r in $\mathcal{P}^I$, all body literals are true w.r.t. I (by construction of P^I), and $H(r) \cap I \neq \emptyset$ (since I is a model of $\mathcal{P}^I$). We differentiate two cases: (i) $H(r) \cap (I \mathbin{\dot\cup} \neg.X) \neq \emptyset$ and (ii) $H(r) \cap (I \mathbin{\dot\cup} \neg.X) = \emptyset$. For (i), r is trivially satisfied by $I \mathbin{\dot\cup} \neg.X$. For (ii), since we know $H(r) \cap I \neq \emptyset$, $H(r) \cap X \neq \emptyset$ must hold. Since X is an unfounded set w.r.t $\mathcal{P}$ and I (and $r \in \mathcal{P}$), a body literal of r must be false w.r.t. $I \mathbin{\dot\cup} \neg.X$ (note that neither a body literal of r is false w.r.t. I since $r \in \mathcal{P}^I$, nor $(H(r) \setminus X) \cap I \neq \emptyset$ holds, otherwise $H(r) \cap (I \mathbin{\dot\cup} \neg.X) \neq \emptyset$). So r is satisfied also in case (ii). $I \mathbin{\dot\cup} \neg.X$ is therefore a model of $\mathcal{P}^I$, and since $(I \mathbin{\dot\cup} \neg.X)^+ \subset I^+$, I is not a minimal model of $\mathcal{P}^I$, contradicting that I is an answer set of $\mathcal{P}$.

$(\Leftarrow)$: By Thm. 4 if I^- is an unfounded set for $\mathcal{P}$ w.r.t. I, I is a model of $\mathcal{P}$, so it is also a model of $\mathcal{P}^I$. We show by contradiction that it is in fact a minimal model of $\mathcal{P}^I$.

Assume that a total interpretation J, where $J^+ \subset I^+$, is a model of $\mathcal{P}^I$. Since both J and I are total, $J^- \supset I^-$. Again by Thm. 4, J^- is an unfounded set for $\mathcal{P}^I$ w.r.t. J. We can then show that J^- is also an unfounded set for $\mathcal{P}$ w.r.t. I, contradicting the fact that I^- is $GUS_{\mathcal{P}}(I)$. For any rule in $\mathcal{P} \setminus \mathcal{P}^I$, a body literal is false w.r.t. I, so condition 1 of Def. 1 holds. For a rule $r \in \mathcal{P}^I$ such that $H(r) \cap J^- \neq \emptyset$, (a) a body literal of r is false w.r.t. J (note that $J \mathbin{\dot\cup} \neg.J^- = J$) or (b) an atom a in $H(r) \setminus J^-$ is true w.r.t. J. Concerning (a), observe that $I \mathbin{\dot\cup} \neg.J^- = J$ so (a) holds iff a body atom is false w.r.t. $I \mathbin{\dot\cup} \neg.J^-$. Concerning (b), since $J^+ \subset I^+$, atom a is also true w.r.t. I. In total, we have shown that J^- is an unfounded set for $\mathcal{P}$ w.r.t. I, a contradiction to $I^- = GUS_{\mathcal{P}}(I)$. So I is indeed a minimal model of $\mathcal{P}^I$, and hence an answer set of $\mathcal{P}$. □

Since the existence of the GUS implies that the interpretation is unfounded-free, we obtain also:

Corollary 3. *A model I of a program $\mathcal{P}$ is unfounded-free iff I is an answer set of $\mathcal{P}$.*

[4] Note that by Def. 3, the existence of $GUS_{\mathcal{P}}(I)$ implies that I is unfounded-free.

5 Computational Complexity

We will now study the complexity involved with unfounded sets. In particular, we are interested in the question whether a total interpretation is unfounded-free, as by Cor. 3 this notion can be fruitfully used for computing answer sets. Throughout this section, we assume that the truth value of aggregates can be established in polynomial time, which is feasible for all aggregates currently available in ASP systems. If, however, aggregate truth valuation has a higher complexity, the total complexity will increase accordingly.

We first show membership for the full language, and then hardness for a restricted fragment, implying completeness for both languages and anything in between.

Theorem 6. *Given a ground disjunctive logic program $\mathcal{P}$, and a total interpretation I, deciding whether I is unfounded-free w.r.t. $\mathcal{P}$ is in co-NP.*

Proof. The complementary problem (deciding whether I is not unfounded-free) is in NP: Guess $X \subseteq B_{\mathcal{P}}$ and check that 1. X is an unfounded set for $\mathcal{P}$ w.r.t. I, and 2. that $X \cap I \neq \emptyset$. Both 1. and 2. are feasible in polynomial time, assuming that determining the truth value of an aggregate literal can be done in polynomial time. □

Next we show that deciding unfounded-freeness is a hard problem even for a simple class of programs, provided that nonmonotone aggregates may be present.

Theorem 7. *Given a ground nondisjunctive, negation-free logic program $\mathcal{P}$ with arbitrary aggregates, and a total interpretation I, deciding whether I is unfounded-free w.r.t. $\mathcal{P}$ is co-NP-hard.*

Proof. We give a reduction from the problem of unsatisfiability of a propositional 3CNF $\phi = (c_1^1 \vee c_2^1 \vee c_3^1) \wedge \ldots \wedge (c_1^m \vee c_2^m \vee c_3^m)$ where each c_j^i is a literal over one of n variables $V = \{x_1, \ldots, x_n\}$. We construct a program $\mathcal{P}(\phi)$:

$$
\begin{array}{lll}
x_1(1) :- \#\mathtt{avg}\{X : x_1(X)\} = 0. & x_1(1) :- w. & x_1(0).\ \ldots\ x_n(0). \\
x_1(-1) :- \#\mathtt{avg}\{X : x_1(X)\} = 0. & x_1(-1) :- w. & w :- \rho(c_1^1), \rho(c_2^1), \rho(c_3^1). \\
\vdots & \vdots & \vdots \\
x_n(1) :- \#\mathtt{avg}\{X : x_n(X)\} = 0. & x_n(1) :- w. & w :- \rho(c_1^m), \rho(c_2^m), \rho(c_3^m). \\
x_n(-1) :- \#\mathtt{avg}\{X : x_n(X)\} = 0. & x_n(-1) :- w. &
\end{array}
$$

where $\rho(x_i) = x_i(1)$ and $\rho(\neg x_i) = x_i(-1)$. Then ϕ is unsatisfiable iff the interpretation $I(\phi) = \{w, x_1(1), x_1(0), x_1(-1), \ldots, x_n(1), x_n(0), x_n(-1)\}$ is unfounded-free.

Indeed, if σ is a satisfying truth assignment for V, then $X^\sigma = \{w\} \cup \{x_i(1) \mid x_i \text{ true in } \sigma\} \cup \{x_i(-1) \mid x_i \text{ false in } \sigma\}$ is unfounded for $\mathcal{P}(\phi)$ w.r.t. $I(\phi)$. It is easily checked that for each rule in $\mathcal{P}(\phi)$ with a head in X^σ at least one body literal is false w.r.t. $I(\phi) \dot\cup \neg.X^\sigma$.

On the other hand, let X be a non-empty unfounded set for $\mathcal{P}(\phi)$ w.r.t. $I(\phi)$. Clearly, $x_i(0) \notin X$. If $x_i(1) \in X$, then $x_i(-1) \notin X$ and vice versa, because if both $x_i(1) \in X$ and $x_i(-1) \in X$, $\#\mathtt{avg}\{X : x_i(X)\} = 0$ is true w.r.t. $I(\phi)$ and $I(\phi) \dot\cup \neg.X$. Furthermore, if some $x_i(1) \in X$ or $x_i(-1) \in X$, then also $w \in X$. If $w \in X$, then for each clause in ϕ some corresponding $x_j(1)$ or $x_j(-1)$ must be in X. It is easy to see that

this corresponds to a (possibly) partial truth assignment satisfying ϕ, which can always be extended to a total truth assignment satisfying ϕ. So any non-empty unfounded set X (hence $X \cap I(\phi) \neq \emptyset$) implies the existence of a satisfying truth assignment for ϕ. □

These results allow us to give a complete picture of the complexity of model checking, reported on the left of Table 1. There, the rows indicate the kinds of aggregates (m – monotone, a – antimonotone, n – nonmonotone) allowed in programs, while the columns vary over the presence of negation and disjunction. All co-NP entries are completeness results. The results in the first row are well-known results of the literature (cf. [19]), the P entries for $\{m, a\}$ follow from recent results in [10], while the other results are consequences of Thms. 5, 6, 7, Cor. 3, with results from the literature.

It becomes clear from this table that a complexity increase occurs with the presence of either disjunction or nonmonotone aggregates, and, importantly, that these two factors together do not cause a further increase. Also in Table 1, on the right hand side, we have summarized results from the literature (see [19,8,10]) for the problem of cautious reasoning. We observe that the complexity increase occurs at the same places, and indeed one can blame the necessity of co-NP checks for the Π_2^P results.

Concerning computation, we conjecture that, given the symmetries in properties and complexity, techniques analogous to those described in [11] can be used in order to effectively and efficiently compute answer sets of programs with arbitrary aggregates.

We will briefly discuss an important issue concerning computation, though. It is striking that from Table 1 it appears that a single, apparently "innocent", aggregate like $\#\texttt{count}\{<1:a>, <2:b>\} = 1$ will increase the reasoning complexity. Obviously, this is not the case, as this aggregate can be rewritten to an equivalent conjunction $\#\texttt{count}\{<1:a>, <2:b>\} \leq 1$, $\#\texttt{count}\{<1:a>, <2:b>\} \geq 1$, thus eliminating the nonmonotone aggregate. In fact, such a decomposition is possible for each nonmonotone aggregate, if one allows the use of custom aggregates (rather than a set of fixed aggregates) and the introduction of new symbols. However, this operation is only polynomial (and hence effective) if the number of the truth value changes of the nonmonotone aggregate in the lattice of total interpretations induced by $<$ is polynomially bounded. Note that all currently implemented nonmonotone aggregates of DLV (#avg is not) and Smodels are polynomially decomposable.

6 Related Work

To our knowledge, the only other works in which the notion of unfounded set has been defined and studied for programs with aggregates are [1,10]. However, both works consider only nondisjunctive programs, and the latter restricts itself to monotone and antimonotone aggregates. As discussed in [10], the definition of [1] seems to ignore aggregates at crucial points of the definition, and appears to be incomparable with the one in [10], and therefore also with Def. 1. Unfounded sets for disjunctive (aggregate-free) programs had been defined and studied in [11]. In fact, several of our results parallel those of [11]. We believe that for this reason the computational techniques reported therein can be adapted to the aggregate setting.

Since unfounded sets have originally been used for defining the well-founded semantics, one could do this also with our unfounded sets. This was done (to some ex-

Table 1. Complexity of Answer Set Checking (left) and Cautious Reasoning (right)

Checking	$\emptyset$	$\{\texttt{not}\}$	$\{\vee\}$	$\{\texttt{not}, \vee\}$
$\emptyset$	P	P	co-NP	co-NP
$\{m, a\}$	P	P	co-NP	co-NP
$\{m, a, n\}$	co-NP	co-NP	co-NP	co-NP

Cautious	$\emptyset$	$\{\texttt{not}\}$	$\{\vee\}$	$\{\texttt{not}, \vee\}$
$\emptyset$	P	co-NP	Π_2^P	Π_2^P
$\{m, a\}$	co-NP	co-NP	Π_2^P	Π_2^P
$\{m, a, n\}$	Π_2^P	Π_2^P	Π_2^P	Π_2^P

tent) in [11], but the recent work in [20] argues that for disjunctive programs a somewhat refined version of unfounded sets based on so-called model sets rather than interpretations. Recently, a unifying framework for unfounded sets and loop formulas has been defined in in [15]. Also this work does not consider aggregates, but we believe that the results with aggregates should be generalizable in a similar way.

Concerning semantics for programs with aggregates, especially the last few years have seen many proposals. We refer to [8] (the definition on which our work is based) and [21] for overviews and comparisons.

7 Conclusion and Future Work

The semantics of logic programs with aggregates is not straightforward, especially in presence of recursive aggregates. The characterizations of answer sets, provided in Sec. 4, allow for a better understanding of the meaning of programs with aggregates. Our results give confidence in the appropriateness of answer sets as defined in [8].

Furthermore, our results provide a handle on effective methods for computing answer sets for disjunctive programs with (possibly recursive and nonmonotone) aggregates. An approach with a separation of model generation and model checking (which is co-NP in the worst case) is indicated by the complexity results of Sec. 5. By defining suitable operators analogously to [11], one can obtain powerful means for pruning in the generation phase, along with an effective instrument for model checking, as described in [13,14]. Our results should also be adaptable to be used for SAT-based ASP systems, all of which rely on loop formulas, along the lines described in [15].

Our complexity results provide a clear picture of the various program fragments from the computational viewpoint. This is very useful for picking the appropriate techniques to be employed for the computation. In particular, it became clear that in the presence of only monotone and antimonotone aggregates and absence of disjunctions, an NP computing scheme can be chosen. That is, there the focus should be on answer set generation, while answer set checking is a simpler task. As soon as nonmonotone aggregates or disjunctions are present, a two-level schema has to be employed, which must focus on both answer set generation and checking, as both tasks are hard. Importantly, the presence of both nonmonotone aggregates and disjunction does not further raise the complexity. It should be noted that, as pointed out at the end of Sec. 5, many nonmonotone aggregates can be decomposed into monotone and antimonotone ones, including Smodels cardinality and weight constraints with positive weights.

A main concern for future work is therefore the exploitation of our results for the implementation of recursive aggregates in ASP systems, along with a study on how to generalize the notion for defining the well-founded semantics for the full language.

References

1. Kemp, D.B., Stuckey, P.J.: Semantics of Logic Programs with Aggregates. In: ISLP'91, MIT Press (1991) 387–401
2. Denecker, M., Pelov, N., Bruynooghe, M.: Ultimate Well-Founded and Stable Model Semantics for Logic Programs with Aggregates. In Codognet, P., ed.: ICLP-2001, (2001) 212–226
3. Gelfond, M.: Representing Knowledge in A-Prolog. In: Computational Logic. Logic Programming and Beyond. LNCS 2408 (2002) 413–451
4. Simons, P., Niemelä, I., Soininen, T.: Extending and Implementing the Stable Model Semantics. Artificial Intelligence **138** (2002) 181–234
5. Dell'Armi, T., Faber, W., Ielpa, G., Leone, N., Pfeifer, G.: Aggregate Functions in DLV. In: ASP'03, Messina, Italy (2003) 274–288 Online at `http://CEUR-WS.org/Vol-78/`.
6. Pelov, N., Truszczyński, M.: Semantics of disjunctive programs with monotone aggregates - an operator-based approach. In: NMR 2004. (2004) 327–334
7. Pelov, N., Denecker, M., Bruynooghe, M.: Partial stable models for logic programs with aggregates. In: LPNMR-7. LNCS 2923
8. Faber, W., Leone, N., Pfeifer, G.: Recursive aggregates in disjunctive logic programs: Semantics and complexity. In: JELIA 2004. LNCS 3229
9. Ferraris, P.: Answer Sets for Propositional Theories. `http://www.cs.utexas.edu/users/otto/papers/proptheories.ps` (2004)
10. Calimeri, F., Faber, W., Leone, N., Perri, S.: Declarative and Computational Properties of Logic Programs with Aggregates. In: Nineteenth International Joint Conference on Artificial Intelligence (IJCAI-05). (2005) To appear.
11. Leone, N., Rullo, P., Scarcello, F.: Disjunctive Stable Models: Unfounded Sets, Fixpoint Semantics and Computation. Information and Computation **135** (1997) 69–112
12. Calimeri, F., Faber, W., Leone, N., Pfeifer, G.: Pruning Operators for Answer Set Programming Systems. In: NMR'2002. (2002) 200–209
13. Koch, C., Leone, N., Pfeifer, G.: Enhancing Disjunctive Logic Programming Systems by SAT Checkers. Artificial Intelligence **15** (2003) 177–212
14. Pfeifer, G.: Improving the Model Generation/Checking Interplay to Enhance the Evaluation of Disjunctive Programs. In: LPNMR-7. LNCS, (2004) 220–233
15. Lee, J.: A Model-Theoretic Counterpart of Loop Formulas. `http://www.cs.utexas.edu/users/appsmurf/papers/mtclf.pdf` (2004)
16. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. CUP (2002)
17. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. NGC **9** (1991) 365–385
18. Van Gelder, A., Ross, K., Schlipf, J.: The Well-Founded Semantics for General Logic Programs. JACM **38** (1991) 620–650
19. Dantsin, E., Eiter, T., Gottlob, G., Voronkov, A.: Complexity and Expressive Power of Logic Programming. ACM Computing Surveys **33** (2001) 374–425
20. Wang, K., Zhou, L.: Comparisons and Computation of Well-founded Semantics for Disjunctive Logic Programs. ACM TOCL **6** (2005)
21. Pelov, N.: Semantics of Logic Programs with Aggregates. PhD thesis, Katholieke Universiteit Leuven, Leuven, Belgium (2004)

Loops: Relevant or Redundant?

Martin Gebser and Torsten Schaub

Institut für Informatik, Universität Potsdam, Postfach 900327, D–14439 Potsdam

Abstract. Loops and the corresponding loop formulas play an important role in answer set programming. On the one hand, they are used for guaranteeing correctness and completeness in SAT-based answer set solvers. On the other hand, they can be used by conventional answer set solvers for finding unfounded sets of atoms. Unfortunately, the number of loops is exponential in the worst case. We demonstrate that not all loops are actually needed for answer set computation. Rather, we characterize the subclass of *elementary loops* and show that they are sufficient and necessary for selecting answer sets among the models of a program's completion. Given that elementary loops cannot be distinguished from general ones in atom dependency graphs, we show how the richer graph structure provided by *body-head dependency graphs* can be exploited for this purpose.

1 Introduction

The success of Answer Set Programming (ASP) is largely due to the availability of efficient solvers, e.g. [1,2]. A similar situation is encountered in the area of satisfiability checking (SAT), in which manifold solvers show an impressive performance. This has led to ASP solvers mapping answer set computation to model generation via SAT solvers [3,4,5]. Since the answer sets of a program form a subset of its classical models, however, additional measures must be taken for eliminating models that are no answer sets. To this end, a program is transformed via *Clark's completion* [6]. The models of the resulting completed program are called *supported models*; they are generally still a superset of the program's answer sets. However, supported models coincide with answer sets on *tight* programs, that is, programs having an acyclic positive atom dependency graph [7]. For example, the program $\{p \leftarrow p\}$ is non-tight; it admits a single empty answer set, while its completion, $\{p \equiv p\}$, has two models, $\emptyset$ and $\{p\}$. While early SAT-based ASP solvers [3] reject non-tight programs, the next generation of solvers, e.g. [4,5], exploits the circular structures within the atom dependency graph for handling non-tight programs. As put forward in [4], the idea is to extend a program's completion by *loop formulas* in order to eliminate the supported models that are no answer sets. Loop formulas are generated from *loops*, which are sets of atoms that circularly depend upon each other in a program's atom dependency graph. Unfortunately, a program may yield exponentially many loops in the worst case [8], so that exponentially many loop formulas may be necessary for filtering out the program's answer sets.

We show that not all loops are needed for selecting the answer sets among the models of a program's completion. Rather, we introduce the subclass of *elementary loops*, whose corresponding loop formulas are sufficient for determining the answer sets of a program from its completion. Moreover, elementary loops are essential in the sense that

C. Baral et al. (Eds.): LPNMR 2005, LNAI 3662, pp. 53–65, 2005.
© Springer-Verlag Berlin Heidelberg 2005

generally none on their loop formulas can be omitted without reintroducing undesired supported models. Given that elementary loops cannot be distinguished from general ones in atom dependency graphs, we show how the richer graph structure provided by *body-head dependency graphs* [9] can be exploited for recognizing elementary loops. Body-head dependency graphs extend atom dependency graphs by an explicit representation of rules' bodies. Their richer graph structure allows for identifying elementary loops in an efficient way. Finally, we show that the set of elementary loops lies between the set of $\subseteq$-minimal loops and the set of all loops. As a consequence, there may still be an exponential number of elementary loops, since there may already be an exponential number of $\subseteq$-minimal loops in the worst case. On the other hand, we show that there may also be exponentially fewer elementary loops than general ones in the best case.

The next section provides the background of this paper. In Section 3, we characterize elementary loops and show that they are sufficient and, generally, necessary for capturing answer sets. Section 4 introduces body-head dependency graphs as a device for recognizing elementary loops. In Section 5, we provide lower and upper bounds for programs' elementary loops. We conclude with Section 6.

2 Background

A *logic program* is a finite set of *rules* of form $a \leftarrow b_1, \dots, b_m, \mathit{not}\ c_1, \dots, \mathit{not}\ c_n$ where $a, b_1, \dots, b_m, c_1, \dots, c_n$ are atoms for $m \geq 0$, $n \geq 0$. Given such a rule r, we denote its *head* a by $\mathit{head}(r)$ and its *body* $\{b_1, \dots, b_m, \mathit{not}\ c_1, \dots, \mathit{not}\ c_n\}$ by $\mathit{body}(r)$. Furthermore, we let $\mathit{body}^+(r) = \{b_1, \dots, b_m\}$ and $\mathit{body}^-(r) = \{c_1, \dots, c_n\}$ be the *positive* and *negative body* of r, respectively. The set of bodies in logic program Π is $\mathit{body}(\Pi) = \{\mathit{body}(r) \mid r \in \Pi\}$. The set of atoms appearing in Π is given by $\mathit{atom}(\Pi)$. A logic program Π is *basic*, if $\mathit{body}^-(r) = \emptyset$ for every rule $r \in \Pi$. The smallest set of atoms closed under basic program Π is denoted by $Cn(\Pi)$. The *reduct* of a logic program Π relative to a set X of atoms is the basic program $\Pi^X = \{\mathit{head}(r) \leftarrow \mathit{body}^+(r) \mid r \in \Pi, \mathit{body}^-(r) \cap X = \emptyset\}$. An *answer set* of a logic program Π is a set X of atoms satisfying $X = Cn(\Pi^X)$.

The *Clark completion* of a program can be defined as follows [6]. For a logic program Π and a rule $r \in \Pi$, define

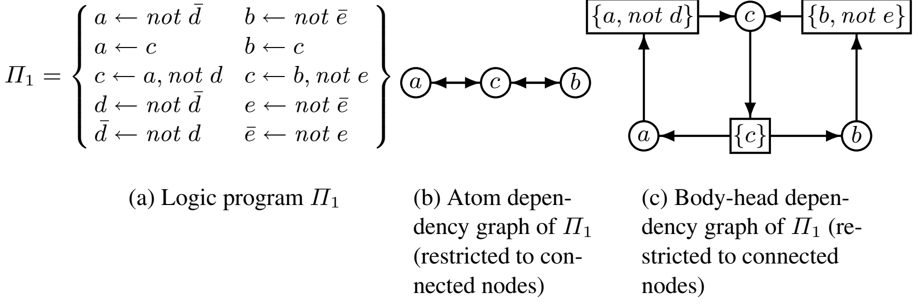
$$\mathit{comp}(r) = \bigwedge\nolimits_{b \in \mathit{body}^+(r)} b \wedge \bigwedge\nolimits_{c \in \mathit{body}^-(r)} \neg c \ ,$$
$$\mathit{comp}(\Pi) = \{a \equiv \bigvee\nolimits_{r \in \Pi, \mathit{head}(r) = a} \mathit{comp}(r) \mid a \in \mathit{atom}(\Pi)\} \ .$$

An answer set of Π is also a model[1] of $\mathit{comp}(\Pi)$. Models of $\mathit{comp}(\Pi)$ are also called *supported models* of Π.

As shown in [4], answer sets can be distinguished among the supported models by means of *loops* in atom dependency graphs (cf. [10,11]). To be precise, the *positive atom dependency graph* of a program Π is the directed graph $(\mathit{atom}(\Pi), E(\Pi))$ where $E(\Pi) = \{(b, a) \mid r \in \Pi, b \in \mathit{body}^+(r), \mathit{head}(r) = a\}$. A set $L \subseteq \mathit{atom}(\Pi)$ is a *loop* in Π, if $(L, E(\Pi, L))$ is a strongly connected subgraph[2] of the positive atom

[1] That is, an interpretation is represented by its entailed set of atoms.

[2] A (sub)graph is strongly connected, if there is a path between any pair of contained nodes.



(a) Logic program Π_1

(b) Atom dependency graph of Π_1 (restricted to connected nodes)

(c) Body-head dependency graph of Π_1 (restricted to connected nodes)

Fig. 1. Logic program Π_1 yielding $loop(\Pi_1) = \{\{a, c\}, \{b, c\}, \{a, b, c\}\}$

dependency graph $(atom(\Pi), E(\Pi))$ such that $E(\Pi, L) = E(\Pi) \cap (L \times L) \neq \emptyset$. Given a loop L in Π, we partition the rules whose heads are in L into two sets, namely

$$R^+(\Pi, L) = \{r \in \Pi \mid head(r) \in L,\ body^+(r) \cap L \neq \emptyset\}\ ,$$
$$R^-(\Pi, L) = \{r \in \Pi \mid head(r) \in L,\ body^+(r) \cap L = \emptyset\}\ .$$

The *loop formula* associated with loop L is

$$LF(\Pi, L) = \neg\big(\bigvee_{r \in R^-(\Pi,L)} comp(r)\big) \rightarrow \bigwedge_{a \in L} \neg a\ . \tag{1}$$

We denote the set of all loops in Π by $loop(\Pi)$. The set of all loop formulas of Π is $LF(\Pi) = \{LF(\Pi, L) \mid L \in loop(\Pi)\}$. As shown in [4], a set X of atoms is an answer set of a logic program Π iff X is a model of $comp(\Pi) \cup LF(\Pi)$.

For illustration, consider Program Π_1 in Figure 1(a). This program has four answer sets: $\{a, b, d, e\}$, $\{b, \bar{d}, e\}$, $\{a, d, \bar{e}\}$, and $\{\bar{d}, \bar{e}\}$. Apart from these, Π_1 has three additional supported models: $\{a, b, c, \bar{d}, e\}$, $\{a, b, c, d, \bar{e}\}$, and $\{a, b, c, \bar{d}, \bar{e}\}$. Observe that each additional supported model is a superset of some answer set. A closer look reveals that all of them contain atoms a, b, and c, which are the ones being involved in loops. In fact, the loops are responsible for the supported models that are no answer sets since they allow for a circular support among atoms. To see this, consider the positive atom dependency graph of Π_1 in Figure 1(b). (We omit atoms d, $\bar{d}$, e, and $\bar{e}$ since they are not involved in any positive dependencies.) We can identify three loops: $\{a, c\}$, $\{b, c\}$, and $\{a, b, c\}$. Each of them induces a strongly connected subgraph that reflects the possibility of circular derivations among these atoms (via rules in $R^+(\Pi_1, \{a, c\})$, $R^+(\Pi_1, \{b, c\})$, and $R^+(\Pi_1, \{a, b, c\})$). This circular behavior can be counterbalanced by the corresponding loop formulas

$$\begin{aligned}
LF(\Pi_1, \{a, c\}) &= \neg(\neg\bar{d} \vee (b \wedge \neg e)) \rightarrow \neg a \wedge \neg c \equiv \bar{d} \wedge (\neg b \vee e) \rightarrow \neg a \wedge \neg c\ ,\\
LF(\Pi_1, \{b, c\}) &= \neg(\neg\bar{e} \vee (a \wedge \neg d)) \rightarrow \neg b \wedge \neg c \equiv \bar{e} \wedge (\neg a \vee d) \rightarrow \neg b \wedge \neg c\ ,\\
LF(\Pi_1, \{a, b, c\}) &= \neg(\neg\bar{d} \vee \neg\bar{e}) \rightarrow \neg a \wedge \neg b \wedge \neg c \equiv \bar{d} \wedge \bar{e} \rightarrow \neg a \wedge \neg b \wedge \neg c\ .
\end{aligned}$$

While these formulas are satisfied by all answer sets of Π_1, one of them is falsified by each of the additional supported models. In this way, $LF(\Pi_1, \{a, c\})$ eliminates $\{a, b, c, \bar{d}, e\}$, $LF(\Pi_1, \{b, c\})$ excludes $\{a, b, c, d, \bar{e}\}$, and $LF(\Pi_1, \{a, b, c\})$ for-

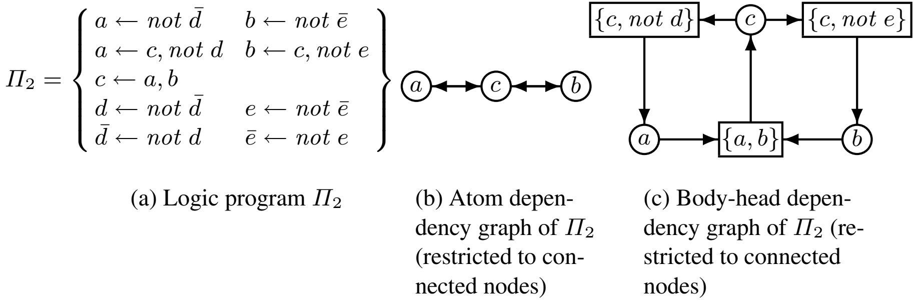


Fig. 2. Logic program Π_2, where $eloop(\Pi_2) = \{\{a, c\}, \{b, c\}\} \subset loop(\Pi_2)$

bids $\{a, b, c, \bar{d}, \bar{e}\}$. Observe that each loop formula prohibits a different supported model and can, thus, not be omitted (although loop $\{a, b, c\}$ contains the two other ones).

3 Elementary Loops

Our main focus lies in characterizing a set of relevant loops, whose loop formulas are *sufficient* and *necessary* for capturing a program's answer sets (together with the completed program). Sufficiency simply means that each model is an answer set. The meaning of necessity is not that straightforward and needs some clarification (see below).

Based on these preliminaries, we introduce the notion of an *elementary loop*.

Definition 1 (Elementary Loop). *Let Π be a logic program and let $L \in loop(\Pi)$.*

We define L as an elementary loop in Π, if, for each loop $L' \in loop(\Pi)$ such that $L' \subset L$,[3] we have $R^-(\Pi, L') \cap R^+(\Pi, L) \neq \emptyset$.

In words, a loop is elementary if each of its strict sub-loops possesses a non-circular support that positively depends on the loop. This characterization is inspired by the structure of loop formulas in (1), according to which non-circular supports form loop formulas' antecedents. If a sub-loop has no non-circular support from the genuine loop, its loop formula's antecedent is satisfied independently. Notably, Section 4 gives a direct characterization of elementary loops that avoids the inspection of sub-loops. As with loops, we denote the set of all elementary loops in a program Π by $eloop(\Pi)$. The set of all elementary loop formulas of Π is denoted by $eLF(\Pi) = \{LF(\Pi, L) \mid L \in eloop(\Pi)\}$. Obviously, we have $eloop(\Pi) \subseteq loop(\Pi)$ and $eLF(\Pi) \subseteq LF(\Pi)$.

Program Π_1 in Figure 1(a) yields the $\subseteq$-minimal loops $\{a, c\}$ and $\{b, c\}$. Such loops are by definition elementary. Moreover, $\{a, b, c\}$ is an elementary loop: Its strict sub-loops, $\{a, c\}$ and $\{b, c\}$, yield $R^-(\Pi_1, \{a, c\}) \cap R^+(\Pi_1, \{a, b, c\}) = \{c \leftarrow b, not\ e\}$ and $R^-(\Pi_1, \{b, c\}) \cap R^+(\Pi_1, \{a, b, c\}) = \{c \leftarrow a, not\ d\}$. The difference between elementary and non-elementary loops shows up when looking at Program Π_2 in Figure 2(a). Similar to Π_1, Program Π_2 has four answer sets: $\{a, b, c, d, e\}$, $\{b, \bar{d}, e\}$, $\{a, d, \bar{e}\}$, and $\{\bar{d}, \bar{e}\}$. Also, both programs share the same positive atom dependency graph, as witnessed by Figures 1(b) and 2(b), respectively. Hence, given that Programs Π_1 and Π_2 are indistinguishable from their positive atom dependency graphs,

[3] We use '$\subset$' to denote the strict subset relation; that is, $L' \subset L$ iff $L' \subseteq L$ and $L' \neq L$.

both programs yield the same set of loops, namely, $loop(\Pi_2) = loop(\Pi_1) = \{\{a,c\}, \{b,c\}, \{a,b,c\}\}$. Unlike this, both programs yield a different set of elementary loops. To see this, observe that for loop $\{a,b,c\}$ and its sub-loops $\{a,c\}$ and $\{b,c\}$, we have

$$R^-(\Pi_2,\{a,c\}) \cap R^+(\Pi_2,\{a,b,c\}) = \{a \leftarrow not\ \bar{d}\} \cap R^+(\Pi_2,\{a,b,c\}) = \emptyset\ ,$$
$$R^-(\Pi_2,\{b,c\}) \cap R^+(\Pi_2,\{a,b,c\}) = \{b \leftarrow not\ \bar{e}\} \cap R^+(\Pi_2,\{a,b,c\}) = \emptyset\ .$$

Thus, $\{a,b,c\}$ is not an elementary loop in Π_2, and $eloop(\Pi_2) = \{\{a,c\},\{b,c\}\}$ is a strict subset of $loop(\Pi_2)$.

As mentioned above, we are interested in a minimal set of essential loops such that their loop formulas in addition to a program's completion capture the program's answer sets. Our next result is a step towards characterizing a sufficient set of loops.

Proposition 2. *Let Π be a logic program and let $L \in loop(\Pi)$ such that $L \notin eloop(\Pi)$.*

Let I be an interpretation of $atom(\Pi)$ such that $L \subseteq I$ and $I \models \neg LF(\Pi, L)$.
Then, there is a loop $L' \in loop(\Pi)$ such that $L' \subset L$ and $I \models \neg LF(\Pi, L')$.

This shows that non-elementary loops are prone to redundancy.

Our first major result is an enhancement of [4, Theorem 1]. That is, elementary loop formulas are, in addition to a program's completion, sufficient for capturing the program's answer sets.

Theorem 3. *Let Π be a logic program and let $X \subseteq atom(\Pi)$.*
Then, X is an answer set of Π iff X is a model of $comp(\Pi) \cup eLF(\Pi)$.

Let us illustrate the two last results by Π_2 in Figure 2(a). Recall that we have

$$eloop(\Pi_2) = \{\{a,c\},\{b,c\}\} \quad \subset \quad \{\{a,c\},\{b,c\},\{a,b,c\}\} = loop(\Pi_2)\ .$$

For Program Π_2, the set $loop(\Pi_2)$ of general loops induces the loop formulas

$$\begin{aligned} LF(\Pi_2,\{a,c\}) &= \neg(\neg\bar{d}) \rightarrow \neg a \wedge \neg c && \equiv \bar{d} \rightarrow \neg a \wedge \neg c\ , \\ LF(\Pi_2,\{b,c\}) &= \neg(\neg\bar{e}) \rightarrow \neg b \wedge \neg c && \equiv \bar{e} \rightarrow \neg b \wedge \neg c\ , \\ LF(\Pi_2,\{a,b,c\}) &= \neg(\neg\bar{d} \vee \neg\bar{e}) \rightarrow \neg a \wedge \neg b \wedge \neg c && \equiv \bar{d} \wedge \bar{e} \rightarrow \neg a \wedge \neg b \wedge \neg c\ . \end{aligned}$$

Observe that

$$LF(\Pi_2,\{a,c\}), LF(\Pi_2,\{b,c\}) \models LF(\Pi_2,\{a,b,c\})\ .$$

That is, loop formula $LF(\Pi_2,\{a,b,c\})$ is redundant and can be removed from $LF(\Pi_2)$ without any risk of producing models of $comp(\Pi_2) \cup (LF(\Pi_2) \setminus \{LF(\Pi_2,\{a,b,c\})\})$ that are no answer sets of Π_2. This outcome is directly obtained when considering elementary loop formulas because $eLF(\Pi_2) = LF(\Pi_2) \setminus \{LF(\Pi_2,\{a,b,c\})\}$.

In what follows, we consider the "necessity" of elementary loops. The problem here is that whether or not a loop formula eliminates unwanted supported models is context dependent because of possible interactions with the completed program and/or among loop formulas. To see this, consider Program $\Pi = \{a \leftarrow ; b \leftarrow a ; b \leftarrow c ; c \leftarrow b\}$. We have $eloop(\Pi) = \{\{b,c\}\}$, but loop formula $LF(\Pi,\{b,c\}) = \neg a \rightarrow \neg b \wedge \neg c$ is

not violated in the single supported model $\{a, b, c\}$ of Π because atom b is supported anyhow by rule $b \leftarrow a$. Furthermore, consider Program $\Pi' = \{a \leftarrow not\ b\,; b \leftarrow not\ a\,; c \leftarrow d, not\ a, not\ b\,; d \leftarrow c, not\ a, not\ b\}$ having elementary loop $\{c, d\}$. The supported models of Π' are $\{a\}$ and $\{b\}$ such that $LF(\Pi', \{c, d\}) = \top \rightarrow \neg c \wedge \neg d$ is not needed for inhibiting circular support among atoms c and d.

In order to capture elementary loops that really produce unwanted supported models, we introduce the notion of an *active elementary loop*.

Definition 4 (Active Elementary Loop). *Let Π be a logic program and let I be an interpretation of $atom(\Pi)$.*

We define $L \in eloop(\Pi)$ as an active elementary loop with respect to I, if

1. *for each rule $r \in R^-(\Pi, L)$, we have $I \models \neg comp(r)$, and*
2. *L is an elementary loop in $\Pi \setminus \{r \in \Pi \mid I \models \neg comp(r)\}$.*

By Condition *1.* an active elementary loop is not non-circularly supported. Condition *2.* ensures that an active elementary loop is still elementary with respect to the rules satisfied by an interpretation; i.e. the rules connecting the elementary loop are not falsified.

The distinguishing property of elementary loops that are active with respect to an interpretation I, as opposed to general loops, is that I "automatically" satisfies the loop formula of any of their sub-loops.

Theorem 5. *Let Π be a logic program, let $L \in eloop(\Pi)$, and let I be an interpretation of $atom(\Pi)$ such that L is active with respect to I.*

Then, we have $I \models \neg LF(\Pi, L)$, and, for each loop $L' \in loop(\Pi)$ such that $L' \subset L$, we have $I \models LF(\Pi, L')$.

For illustration, reconsider Programs Π_1 and Π_2 (cf. Figures 1(a) and 2(a)). Both programs yield the loops $\{a, c\}$, $\{b, c\}$, and $\{a, b, c\}$. The difference between Π_1 and Π_2 is that $\{a, b, c\}$ is an elementary loop in Π_1, but not in Π_2. For Π_1, this means that, if $\{a, b, c\}$ is active with respect to a supported model M of $comp(\Pi_1)$, M is also model of $comp(\Pi_1) \cup eLF(\Pi_1) \setminus \{LF(\Pi_1, \{a, b, c\})\}$. In fact, the supported model $M = \{a, b, c, \bar{d}, \bar{e}\}$ violates

$$LF(\Pi_1, \{a, b, c\}) = \neg(\neg\bar{d} \vee \neg\bar{e}) \rightarrow \neg a \wedge \neg b \wedge \neg c \equiv \bar{d} \wedge \bar{e} \rightarrow \neg a \wedge \neg b \wedge \neg c$$

but satisfies

$$\begin{aligned} LF(\Pi_1, \{a, c\}) &= \neg(\neg\bar{d} \vee (b \wedge \neg e)) \rightarrow \neg a \wedge \neg c \equiv \bar{d} \wedge (\neg b \vee e) \rightarrow \neg a \wedge \neg c\,, \\ LF(\Pi_1, \{b, c\}) &= \neg(\neg\bar{e} \vee (a \wedge \neg d)) \rightarrow \neg b \wedge \neg c \equiv \bar{e} \wedge (\neg a \vee d) \rightarrow \neg b \wedge \neg c\,. \end{aligned}$$

Hence, we cannot skip $LF(\Pi_1, \{a, b, c\})$ without producing a supported model that is no answer set. In contrast to this, no model of $comp(\Pi_2)$ violates $LF(\Pi_2, \{a, b, c\})$ and satisfies both $LF(\Pi_2, \{a, c\})$ and $LF(\Pi_2, \{b, c\})$. This follows directly from Theorem 3, as $\{a, b, c\}$ is not an elementary loop in Π_2.

4 Graph-Theoretical Characterization of Elementary Loops

We have seen in the previous section that elementary and non-elementary loops cannot be distinguished using atom dependency graphs (cf. Figures 1(b) and 2(b)). Furthermore, Definition 1 suggests examining all strict sub-loops for finding out whether a

loop is elementary. This is intractable as a loop may have exponentially many strict sub-loops. In what follows, we show that identifying elementary loops can be done efficiently based on a refined concept of a dependency graph.

First of all, we introduce the *body-head dependency graph* of a program.

Definition 6 (Positive Body-Head Dependency Graph). *Let Π be a logic program.*

We define the positive body-head dependency graph of Π as the directed graph $(atom(\Pi) \cup body(\Pi), E_0(\Pi) \cup E_2(\Pi))$ where

$$E_0(\Pi) = \{(b, B) \mid r \in \Pi,\ b \in body^+(r),\ body(r) = B\}\ ,$$
$$E_2(\Pi) = \{(B, a) \mid r \in \Pi,\ body(r) = B,\ head(r) = a\}\ .$$

Body-head dependency graphs were introduced in [9] as a formal device for characterizing answer set computation. In fact, fully-fledged body-head dependency graphs constitute the primary data structure of the `nomore` answer set solver [12]. In addition to the edges in $E_0(\Pi)$ and $E_2(\Pi)$, they contain a type of edges for negative dependencies, namely, $E_1(\Pi) = \{(c, B) \mid r \in \Pi,\ c \in body^-(r),\ body(r) = B\}$.[4] In what follows, we often drop the attribute 'positive' and simply write body-head or atom dependency graph, respectively, since loops exclusively rely on positive dependencies.

Definition 7 (Induced Subgraph). *Let Π be a logic program and let $A \subseteq atom(\Pi)$.*

We define the induced subgraph of A in Π as the directed graph $(A \cup body(\Pi, A), E_0(\Pi, A) \cup E_2(\Pi, A))$ where

$$body(\Pi, A) = \{B \in body(\Pi) \mid b \in A,\ (b, B) \in E_0(\Pi),\ a \in A,\ (B, a) \in E_2(\Pi)\}\ ,$$
$$E_0(\Pi, A) = E_0(\Pi) \cap (A \times body(\Pi, A))\ ,$$
$$E_2(\Pi, A) = E_2(\Pi) \cap (body(\Pi, A) \times A)\ .$$

Note that, in the induced subgraph of a set A of atoms, we only include those bodies that contain an atom in A and that also occur in a rule whose head is in A. That is, the bodies, which are responsible for edges in atom dependency graphs, are made explicit in body-head dependency graphs as nodes in-between atoms.

Figure 1(c) shows the body-head dependency graph of Π_1. As in Figure 1(b), we leave out isolated nodes, that is here, purely negative bodies and atoms not occurring in positive bodies. Unlike this, atom a is contained in the graph since it occurs in the positive body of rule $c \leftarrow a, not\ d$; accordingly, the edge $(a, \{a, not\ d\})$ belongs to the set of edges $E_0(\Pi_1)$ of the body-head dependency graph. Among the edges in $E_2(\Pi_1)$, we find $(\{a, not\ d\}, c)$ because of rule $c \leftarrow a, not\ d$. The induced subgraph of $\{a, c\}$ in Π_1 contains atoms a and c, bodies $\{a, not\ d\}$ and $\{c\}$, and their connecting edges.

As with atom dependency graphs, a set of atoms is a loop if its induced subgraph is a non-trivial strongly connected graph.

Proposition 8. *Let Π be a logic program and let $L \subseteq atom(\Pi)$.*

L is a loop in Π iff the induced subgraph of L in Π is a strongly connected graph such that $body(\Pi, L) \neq \emptyset$.

[4] The notation traces back to [13]; the sum of labels in a cycle indicates whether the cycle is even or odd.

In order to describe elementary loops graph-theoretically, we introduce the *elementary subgraph* of a set of atoms, which is more fine grained than the induced subgraph.

Definition 9 (Elementary Subgraph). *Let Π be a logic program, let $A \subseteq atom(\Pi)$, and let $(A \cup body(\Pi, A), E_0(\Pi, A) \cup E_2(\Pi, A))$ be the induced subgraph of A in Π.*

We define the elementary closure of A in Π as the set $eCl(\Pi, A)$ of edges where

$$\begin{aligned} eCl^0(\Pi, A) &= \emptyset\,, \\ eCl^{i+1}(\Pi, A) &= eCl^i(\Pi, A) \cup \{(b, B) \in E_0(\Pi, A) \mid \textit{there is} \\ &\qquad \textit{a path in } (A \cup body(\Pi, A), eCl^i(\Pi, A) \cup E_2(\Pi, A)) \\ &\qquad \textit{from } b \in A \textit{ to each } b' \in A \textit{ such that } (b', B) \in E_0(\Pi, A)\}\,,^5 \\ eCl(\Pi, A) &= \textstyle\bigcup_{i \in I\!N} eCl^i(\Pi, A)\,. \end{aligned}$$

We define the elementary subgraph of A in Π as the directed graph $(A \cup body(\Pi, A), eCl(\Pi, A) \cup E_2(\Pi, A))$.

The general purpose of elementary subgraphs is to distinguish essential from superfluous dependencies. Let us illustrate this by rule $c \leftarrow a, b$ in Π_2 and consider the body-head dependency graph of Π_2 in Figure 2(c). Here, atom c positively depends on atoms a and b through body $\{a, b\}$. In Π_2, c is unfounded if either a or b and c itself are not non-circularly supported; that is, the other atom cannot help in non-circularly supporting a and c or b and c, respectively. The situation changes if a and b take part in a loop independently from c. Then, a and b non-circularly support c if there is a non-circular support for either a or b. The elementary closure reflects these issues by stipulating that there is already a path from one to the other predecessors of a body before an edge to the body can be added. This allows for distinguishing essential dependencies from superfluous ones.

Our next major result shows that elementary subgraphs make the difference between elementary loops and non-elementary ones.

Theorem 10. *Let Π be a logic program and let $L \subseteq atom(\Pi)$.*

L is an elementary loop in Π iff the elementary subgraph of L in Π is a strongly connected graph such that $body(\Pi, L) \neq \emptyset$.

For illustrating the previous result, reconsider Figure 1(c) showing the connected part of the body-head dependency graph of Program Π_1. Observe that each contained body is reached by precisely one edge. Therefore, we have $eCl^1(\Pi_1, A) = E_0(\Pi_1, A)$ for every $A \subseteq atom(\Pi_1)$, and elementary subgraphs coincide with induced subgraphs.

The body-head dependency graph of Program Π_2 is different from the one of Program Π_1, as witnessed by Figures 1(c) and 2(c). In Figure 2(c), we see the connected part of the body-head dependency graph of Π_2, which coincides with the induced subgraph of loop $\{a, b, c\}$ in Π_2. Regarding the elementary closure of $\{a, b, c\}$, we have

$$eCl(\Pi_2, \{a, b, c\}) = eCl^1(\Pi_2, \{a, b, c\}) = \{(c, \{c, not\ d\}), (c, \{c, not\ e\})\}\,.$$

[5] Note that the path from b to b' can be trivial, i.e. $b = b'$.

Observe that $eCl(\Pi_2, \{a, b, c\})$ does not contain edges $(a, \{a, b\})$ and $(b, \{a, b\})$. This is because a as well as b must have a path to the other atom before the respective edge can be added to the elementary closure. Since there are no such paths, none of the edges can ever be added. As a consequence, atoms a and b have no outgoing edges in the elementary subgraph of $\{a, b, c\}$ in Π_2, which is not strongly connected. This agrees with the observation made in Section 3 that $\{a, b, c\}$ is not an elementary loop in Π_2. In contrast to $\{a, b, c\}$, the elementary subgraphs of loops $\{a, c\}$ and $\{b, c\}$ in Π_2 are strongly connected, verifying $eloop(\Pi_2) = \{\{a, c\}, \{b, c\}\}$.

As observed on Program Π_1, elementary subgraphs coincide with induced subgraphs on unary programs, having at most one positive body atom. For such programs, every general loop is also an elementary one.

Proposition 11. *Let Π be a logic program such that $|body^+(r)| \leq 1$ for all $r \in \Pi$. Then, we have $eloop(\Pi) = loop(\Pi)$.*

Note that unary programs are strictly less expressive than general ones, as shown in [14].

The analysis of elementary subgraphs yields that each contained atom must be the unique predecessor of some body; otherwise, the atom has no outgoing edge in the elementary closure. Moreover, the induced subgraph of an elementary loop cannot be torn apart by removing edges to bodies, provided that each body is still reachable.

Proposition 12. *Let Π be a logic program and let $L \in eloop(\Pi)$. Then, the induced subgraph of L in Π, $(L \cup body(\Pi, L), E_0(\Pi, L) \cup E_2(\Pi, L))$, has the following properties:*

1. *For each atom $b \in L$, there is a body $B \in body(\Pi, L)$ such that $\{b\} = \{b' \in L \mid (b', B) \in E_0(\Pi, L)\}$.*
2. *For every set $E_0^{\subseteq} \subseteq E_0(\Pi, L)$ of edges such that $\{B \in body(\Pi, L) \mid b \in L, (b, B) \in E_0^{\subseteq}\} = body(\Pi, L)$, we have that $(L \cup body(\Pi, L), E_0^{\subseteq} \cup E_2(\Pi, L))$ is a strongly connected graph.*

Although we refrain from giving a specific algorithm, let us note that the concept of elementary subgraphs allows for computing elementary loops efficiently by means of standard graph algorithms. In particular, deciding whether a set of atoms is an elementary loop can be done in linear time.

5 Elementary Versus Non-elementary Loops

This section compares the sets of a program's elementary and general loops. By Theorem 3, loop formulas for non-elementary loops need not be added to a program's completion in order to capture the program's answer sets. With this information at hand, we are interested in how many loop formulas can be omitted in the best or in the worst case, respectively.

First, we determine a lower bound on the set of a program's elementary loops. Such a bound is immediately obtained from Definition 1, because a loop is trivially elementary if it has no strict sub-loops. Thus, we have $mloop(\Pi) \subseteq eloop(\Pi)$ where $mloop(\Pi)$ denotes the set of $\subseteq$-minimal loops in a program Π. Second, the set of a

program's loops constitutes an upper bound for the program's elementary loops, also by Definition 1. Thus, we have $eloop(\Pi) \subseteq loop(\Pi)$. Finally, the question is how the set of a program's loops can be bound from above. In order to answer it, we define the set of loops that are $\subseteq$-minimal for an atom $a \in atom(\Pi)$ as $aloop(\Pi, a) = \{L \in loop(\Pi) \mid a \in L,$ *there is no loop* $L' \in loop(\Pi)$ *such that* $a \in L'$ *and* $L' \subset L\}$. For a program Π, we let $aloop(\Pi) = \bigcup_{a \in atom(\Pi)} aloop(\Pi, a)$.

In the worst case, any non-empty combination of loops in $aloop(\Pi)$ is a loop, and we obtain the following upper bound for a program's loops.

Proposition 13. *Let Π be a logic program.*
Then, we have $loop(\Pi) \subseteq \{\bigcup_{L \in A} L \mid A \in 2^{aloop(\Pi)} \setminus \{\emptyset\}\}$.

Taking the above considerations together, we obtain the following estimation.

Corollary 14. *Let Π be a logic program. Then, we have*

$$mloop(\Pi) \subseteq eloop(\Pi) \subseteq loop(\Pi) \subseteq \{\bigcup_{L \in A} L \mid A \in 2^{aloop(\Pi)} \setminus \{\emptyset\}\} .$$

In what follows, we give some schematic examples with programs Π for which $loop(\Pi) = \{\bigcup_{L \in A} L \mid A \in 2^{aloop(\Pi)} \setminus \{\emptyset\}\}$. Our first program sketches the worst case, i.e. $eloop(\Pi) = \{\bigcup_{L \in A} L \mid A \in 2^{aloop(\Pi)} \setminus \{\emptyset\}\}$, whereas the second program reflects the best case that $eloop(\Pi) = mloop(\Pi)$. The programs show that the set of elementary loops can vary significantly between the given lower and the upper bound.

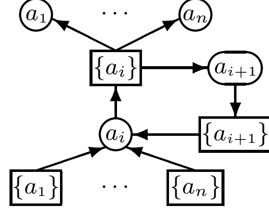
For illustrating the worst case, consider Program Π_3 in Figure 3(a). First observe that $|body^+(r)| = 1$ for every rule $r \in \Pi_3$. Thus by Proposition 11, each loop in Π_3 is elementary. The atom dependency graph of Π_3 is a complete graph because there is a rule $a_i \leftarrow a_j$ for every pair of distinct atoms $a_i \in atom(\Pi_3), a_j \in atom(\Pi_3)$. As a consequence, any combination of distinct elementary loops gives a new elementary loop, and we have $eloop(\Pi_3) = \{\bigcup_{L \in A} L \mid A \in 2^{aloop(\Pi_3)} \setminus \{\emptyset\}\}$.

Program Π_4 in Figure 3(c) is complementary to Π_3. Here $|body^+(r)| = |atom(\Pi_4)| - 1$ for every rule $r \in \Pi_4$. However, the atom dependency graph of Π_4 is identical to that of Π_3. As observed with Π_1 and Π_2 (cf. Figures 1(a) and 2(b)), Π_3 and Π_4 are thus indistinguishable from their atom dependency graphs. Again the body-head dependency graphs reveal the different natures of Π_3 and Π_4. We have that, similar to Π_3, every two-elementary subset of $atom(\Pi_4)$ forms a $\subseteq$-minimal and, thus, elementary loop. However, looking at the body-head dependency graph of Π_4 in Figure 3(d), we see that each atom has a single body as predecessor (i.e. there is a single supporting rule) such that distinct elementary loops can only be "glued" at bodies. In the resulting induced subgraph, bodies have several predecessors. Such an induced subgraph does not satisfy property *1.* from Proposition 12, and the obtained loop is non-elementary. Thus, we have $eloop(\Pi_4) = mloop(\Pi_4)$ and can omit loop formulas for all loops in $\{\bigcup_{L \in A} L \mid A \in 2^{aloop(\Pi_4)} \setminus \{\emptyset\}\} \setminus mloop(\Pi_4)$.

The achievements obtainable through using elementary instead of general loops can be underpinned by looking at the approaches of `assat` [4] and `smodels` [1] for dealing with unfounded sets. The `assat` system is based on a program's completion and identifies loops, whose loop formulas are violated, on demand, that is, whenever a

$$\Pi_3 = \left\{ \begin{array}{lll} a_1 \leftarrow a_2 & \dots & a_1 \leftarrow a_n \\ \vdots & & \\ a_i \leftarrow a_1 & \dots & a_i \leftarrow a_n \\ a_{i+1} \leftarrow a_1 & \dots & a_{i+1} \leftarrow a_n \\ \vdots & & \\ a_n \leftarrow a_1 & \dots & a_n \leftarrow a_{n-1} \end{array} \right\}$$

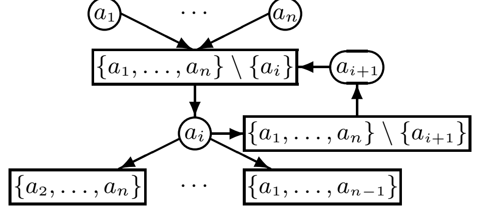
(a) Schematic program Π_3



(b) Schematic body-head dependency graph of Π_3

$$\Pi_4 = \left\{ \begin{array}{l} a_1 \leftarrow a_2, \dots, a_n \\ \vdots \\ a_i \leftarrow a_1, \dots, a_{i-1}, a_{i+1}, \dots, a_n \\ a_{i+1} \leftarrow a_1, \dots, a_i, a_{i+2}, \dots, a_n \\ \vdots \\ a_n \leftarrow a_1, \dots, a_{n-1} \end{array} \right\}$$

(c) Schematic program Π_4



(d) Schematic body-head dependency graph of Π_4

Fig. 3. Logic programs Π_3 and Π_4, where $loop(\Pi_i) = \{\bigcup_{L \in A} L \mid A \in 2^{aloop(\Pi_i)} \setminus \{\emptyset\}\}$, $eloop(\Pi_3) = \{\bigcup_{L \in A} L \mid A \in 2^{aloop(\Pi_3)} \setminus \{\emptyset\}\}$, and $eloop(\Pi_4) = mloop(\Pi_4)$

supported model not representing an answer set has been found. The circular support of such loops is in future prohibited by loop formulas such that an unwanted supported model cannot be recomputed. Now assume that the supported model $\{a, b, c, \bar{d}, \bar{e}\}$ is found first for Program Π_2 in Figure 2(a). Then `assat` identifies $\{a, b, c\}$ as a so-called *terminating loop* [4] and adds loop formula

$$LF(\Pi_2, \{a, b, c\}) = \neg(\neg\bar{d} \vee \neg\bar{e}) \rightarrow \neg a \wedge \neg b \wedge \neg c \equiv \bar{d} \wedge \bar{e} \rightarrow \neg a \wedge \neg b \wedge \neg c$$

to $comp(\Pi_2)$ before searching for another supported model. The problem is that loop $\{a, b, c\}$ is non-elementary and that circular support within elementary loops $\{a, c\}$ and $\{b, c\}$ is not prohibited by $LF(\Pi_2, \{a, b, c\})$. Consequently, `assat` may find supported models $\{a, b, c, \bar{d}, e\}$ and $\{a, b, c, d, \bar{e}\}$ next, necessitating additional loop formulas

$$LF(\Pi_2, \{a, c\}) = \neg(\neg\bar{d}) \rightarrow \neg a \wedge \neg c \equiv \bar{d} \rightarrow \neg a \wedge \neg c \quad \text{and}$$
$$LF(\Pi_2, \{b, c\}) = \neg(\neg\bar{e}) \rightarrow \neg b \wedge \neg c \equiv \bar{e} \rightarrow \neg b \wedge \neg c \,,$$

before finding the first answer set. The possibility of computing the supported models $\{a, b, c, \bar{d}, e\}$ and $\{a, b, c, d, \bar{e}\}$ can be avoided by splitting the non-elementary loop $\{a, b, c\}$ into its elementary sub-loops $\{a, c\}$ and $\{b, c\}$. Besides $\{a, c\}$ and $\{b, c\}$, $LF(\Pi_2, \{a, c\})$ and $LF(\Pi_2, \{b, c\})$ prohibit circular support within loop $\{a, b, c\}$, and `assat` may treat three loops using only two loop formulas. In general, the elementary closure, as given in Definition 9, can be used for checking whether a terminating loop is

elementary. If not, an elementary sub-loop, whose loop formula also prohibits circular support within the genuine terminating loop, can be determined.[6]

The smodels answer set solver falsifies greatest unfounded sets in its function *atmost*. At the implementation level, *atmost* is restricted to strongly connected components of a program's positive atom dependency graph (but may spread over different components if an unfounded set is detected) [1]. When *atmost* is applied to Program Π_4 in Figure 3(c) (or a program having a comparable body-head dependency graph), it has to take the whole strongly connected component induced by $atom(\Pi_4)$ into consideration, since the atom dependency graph of Π_4 is complete. The efforts of *atmost* can be restricted by concentrating on elementary loops, which are pairs of atoms in case of Π_4. That is, any pair of unfounded atoms is sufficient for falsifying the bodies of all rules that contribute to the strongly connected component induced by $atom(\Pi_4)$.

Finally, it is noteworthy to mention that [15] describes how the computation of a program's well-founded model [16] simplifies based on certain properties of the program's full atom dependency graph (i.e. both positive and negative edges are included). The simplifications can be applied if a strongly connected component contains either only positive edges or if no atom depends positively on itself. The first case reflects that the contained atoms are involved in a loop and the second that circular support is impossible. An interesting topic for future investigation is whether the above conditions can be refined using the richer structure of body-head dependency graphs, which, for instance, allows for distinguishing between elementary and non-elementary loops.

6 Conclusion

The purpose of loop formulas is to falsify unfounded sets whose atoms circularly depend upon each other in a given program. The detection of unfounded sets traces back to well-founded semantics [16]. Basically, the well-founded semantics infers atoms that are consequences of a program's rules and falsifies unfounded sets. In accord with the well-founded semantics, all atoms in an answer set are consequences and no atom is unfounded. Complementary to [16] concentrating on *greatest* unfounded sets, this paper investigates indispensable unfounded sets whose falsification is essential for answer set computation. To this end, we have introduced the notion of an elementary loop and have described it using body-head dependency graphs. Although we cannot avoid the theoretical barrier of exponentially many loops in the worst case, we have shown that elementary loops provide necessary and sufficient criteria for characterizing answer sets. Apart from their theoretical importance, our results have furthermore a practical impact since they allow to focus the computation in ASP solvers to ultimately necessary parts. An interesting topic for future research will be generalizing our new concept of elementary loops to disjunctive programs, as has been done for general loops in [17].

Acknowledgments. We are grateful to Enno Schultz and the anonymous referees for their helpful comments, leading to a new presentation of our contribution. This work

[6] A non-elementary loop may yield exponentially many elementary sub-loops. Thus, identifying *all* elementary sub-loops might be intractable. However, the number of elementary loops needed to cover the atoms in a (general) terminating loop of size n is bound by n.

was supported by DFG under grant SCHA 550/6-4 as well as the EC through IST-2001-37004 WASP project.

References

1. Simons, P., Niemelä, I., Soininen, T.: Extending and implementing the stable model semantics. Artificial Intelligence **138** (2002) 181–234
2. Leone, N., Faber, W., Pfeifer, G., Eiter, T., Gottlob, G., Koch, C., Mateis, C., Perri, S., Scarcello, F.: The DLV system for knowledge representation and reasoning. ACM Transactions on Computational Logic (2005) To appear.
3. Babovich, Y., Lifschitz, V.: Computing answer sets using program completion. Unpublished draft. (2003)
4. Lin, F., Zhao, Y.: Assat: computing answer sets of a logic program by sat solvers. Artificial Intelligence **157** (2004) 115–137
5. Lierler, Y., Maratea, M.: Cmodels-2: Sat-based answer sets solver enhanced to non-tight programs. In Lifschitz, V., Niemelä, I., eds.: Proceedings of the Seventh International Conference on Logic Programming and Nonmonotonic Reasoning. Springer (2004) 346–350
6. Clark, K.: Negation as failure. In Gallaire, H., Minker, J., eds.: Logic and Data Bases. Plenum Press (1978) 293–322
7. Fages, F.: Consistency of clark's completion and the existence of stable models. Journal of Methods of Logic in Computer Science **1** (1994) 51–60
8. Lifschitz, V., Razborov, A.: Why are there so many loop formulas? ACM Transactions on Computational Logic (To appear.)
9. Linke, T.: Suitable graphs for answer set programming. In Vos, M.D., Provetti, A., eds.: Proceedings of the Second International Workshop on Answer Set Programming. CEUR Workshop Proceedings (2003) 15–28
10. Apt, K., Blair, H., Walker, A.: Towards a theory of declarative knowledge. In Minker, J., ed.: Foundations of Deductive Databases and Logic Programming. Morgan Kaufmann Publishers (1987) 89–148
11. Przymusinski, T.: On the declarative semantics of deductive databases and logic programs. In Minker, J., ed.: Foundations of Deductive Databases and Logic Programming. Morgan Kaufmann Publishers (1988) 193–216
12. (`http://www.cs.uni-potsdam.de/~linke/nomore`)
13. Papadimitriou, C., Sideri, M.: Default theories that always have extensions. Artificial Intelligence **69** (1994) 347–357
14. Janhunen, T.: Comparing the expressive powers of some syntactically restricted classes of logic programs. In Lloyd, J., Dahl, V., Furbach, U., Kerber, M., Lau, K., Palamidessi, C., Pereira, L., Sagiv, Y., Stuckey, P., eds.: Proceedings of the First International Conference on Computational Logic. Springer (2000) 852–866
15. Dix, J., Furbach, U., Niemelä, I.: Nonmonotonic reasoning: Towards efficient calculi and implementations. In Robinson, J., Voronkov, A., eds.: Handbook of Automated Reasoning. Elsevier and MIT Press (2001) 1241–1354
16. van Gelder, A., Ross, K., Schlipf, J.: The well-founded semantics for general logic programs. Journal of the ACM **38** (1991) 620–650
17. Lee, J., Lifschitz, V.: Loop formulas for disjunctive logic programs. In Palamidessi, C., ed.: Proceedings of 19th International Conference on Logic Programming. Springer (2003) 451–465

Approximating Answer Sets of Unitary Lifschitz-Woo Programs

Victor W. Marek[1], Inna Pivkina[2], and Mirosław Truszczyński[1]

[1] Department of Computer Science, University of Kentucky, Lexington, KY 40506-0046, USA
[2] Department of Computer Science, New Mexico State University, P.O. Box 30001, MSC CS, Las Cruces, NM 88003, USA

Abstract. We investigate techniques for approximating answer sets of general logic programs of Lifschitz and Woo, whose rules have single literals as heads. We propose three different methods of approximation and obtain results on the relationship between them. Since general logic programs with single literals as heads are equivalent to revision programs, we obtain results on approximations of justified revisions of databases by revision programs.

1 Introduction

General logic programs were introduced by Lifschitz and Woo [LW92]. Their syntax follows closely that of disjunctive logic programs but there is one essential difference. The operator **not**, representing the *default negation* is no longer confined to the bodies of program rules but may appear in their heads, as well. Lifschitz and Woo [LW92] showed that the semantics of answer sets introduced for disjunctive logic programs in [GL91] can be lifted to the class of general logic programs.

In this paper, we study the class of those general programs that do not contain disjunctions in the heads of their rules. We call such programs *unitary*. Unitary general programs are of interest for two reasons. First, they go beyond the class of normal logic programs by allowing the default-negation operator in the rule heads. Second, in a certain precise sense, unitary general programs are equivalent to the class of revision programs [MT98, MPT02], which provide a formalism for describing and enforcing database revisions. Consequently, results for unitary general programs extend to the case of revision programs.

The problem we focus on in this paper is that of approximating answer sets of unitary general programs. The problem to decide whether a unitary logic program has an answer set is NP-complete[1]. Consequently, computing answer sets of unitary general programs is hard and it is important to establish efficient ways to approximate them. On one hand, such approximations can be sufficient for some reasoning tasks. On the other hand, they can be used by programs computing answer sets to prune the search space and can improve their performance significantly.

[1] Without the restriction to unitary programs (and assuming that the polynomial hierarchy does not collapse) the problem is even harder — Σ_2^P-complete.

C. Baral et al. (Eds.): LPNMR 2005, LNAI 3662, pp. 66–78, 2005.
© Springer-Verlag Berlin Heidelberg 2005

In the case of normal logic programs the well-founded model [VRS88] provides an effective approximation to all answer sets[2]. It can be computed in polynomial time and is known to provide an effective pruning mechanism for programs computing stable models [SNV95, SNS02]. An obvious approach to the problem at hand seems to be then to extend the well-founded model and its properties to the class of unitary programs. However, despite similarities between normal and unitary programs, no counterpart of the well-founded model has been proposed for the latter class so far, and whether it can be done remains unresolved.

Thus, we approach the problem not by attempting to generalize the well-founded semantics but by exploiting this semantics in some other, less direct ways. Namely, we introduce three operators for unitary general programs and use them to define the approximations. The first two operators are antimonotone and are closely related to operators behind the well-founded semantics of normal logic programs. Iterating them yields *alternating* sequences. We use the limits of these sequences to construct our first two approximations to answer sets of unitary general programs. The two approximations we obtain in this way are not comparable (neither is stronger than the other one). The third operator is not antimonotone in general. However, in the case of unitary general programs that have answer sets, iterating this operator results in an alternating sequence and the limit of this sequence yields yet another approximation to answer sets of unitary general programs. We show that this third approximation is stronger than the other two. We also show that all three approaches imply sufficient conditions for the *non-existence* of answer sets of unitary programs.

As we noted, unitary programs are related to revision programs [MT98, MPT99]. Having introduced approximations to answer sets of unitary general programs, we show that our results apply in a direct way to the case of revision programming.

All programs we consider in the paper are *finite*. That assumption simplifies arguments. However, all our results can be extended to the case of infinite programs.

2 Preliminaries

Atoms and literals. In the paper we consider a fixed set U of (propositional) atoms. Expressions of the form a and $\mathbf{not}(a)$, where $a \in U$, are *literals* (over U). We denote the set of all literals over U by $Lit(U)$. A set of literals $L \subseteq Lit(U)$ is *coherent* if there is no $a \in U$ such that both $a \in L$ and $\mathbf{not}(a) \in L$. A set of literals $L \subseteq Lit(U)$ is *complete* if for every $a \in U$, $a \in L$ or $\mathbf{not}(a) \in L$ (it is possible that for some a, both $a \in L$ and $\mathbf{not}(a) \in L$).

For a set M of atoms, $M \subseteq U$, we define

$$\mathbf{not}(M) = \{\mathbf{not}(a)\colon a \in M\} \text{ and } M^c = M \cup \mathbf{not}(U \setminus M).$$

The mapping $M \mapsto M^c$ is a bijection between subsets of U and coherent and complete sets of literals contained in $Lit(U)$.

[2] In the context of normal logic programming, answer sets are more commonly known as *stable models*.

Unitary general programs. A *unitary general logic program*, or *UG-program* is a collection of rules of the form:

$$\alpha \leftarrow \alpha_1, \ldots, \alpha_m \tag{1}$$

where $\alpha, \alpha_1, \ldots, \alpha_m$ are literals from $Lit(U)$. The literal α is the *head* of the rule. The set of literals $\{\alpha_1, \ldots, \alpha_m\}$ is the *body* of the rule.

Let P be a UG-program. We write P^+ (respectively, P^-) to denote programs consisting of all rules in P that have an atom (respectively, a negated atom) as the head.

Satisfaction and models. A set of atoms $M \subseteq U$ *satisfies* (is a *model* of) an atom $a \in U$ (respectively, a literal $\mathbf{not}(a) \in Lit(U)$), if $a \in M$ (respectively, $a \notin M$). The concept of satisfaction (being a model of) extends in a standard way to rules and programs. As usual, we write $\models$ to denote the satisfaction relation.

Sets of literals closed under UG-programs. In addition to models, we also associate with a UG-program P sets of literals that are closed under rules in P. A set L of literals is *closed* under a UG-program P if for every rule $r = \alpha \leftarrow Body \in P$ such that $Body \subseteq L$, $\alpha \in L$. One can show that every UG-program P has a least set of literals closed under its rules[3]. We denote it by P^*. We observe that if P is a definite Horn program, P^* consists of atoms only and coincides with the least model of P.

Stable models of normal logic programs. Models are too weak for knowledge representation applications. In the case of normal logic programs, the appropriate semantic concept is that of a stable model. We recall that according to the original definition [GL88], a set of atoms M is a stable model of a normal logic program P if

$$[P^M]^* = M, \tag{2}$$

where P^M is the *Gelfond-Lifschitz* reduct of P with respect to M. The following characterization of stable models is well known [BTK93]: M is a stable model of a normal logic program P if and only if

$$[P \cup \mathbf{not}(U \setminus M)]^* \cap U = M. \tag{3}$$

Answer sets of UG-programs. Lifschitz and Woo [LW92] extended the concept of a stable model to the case of arbitrary general programs and called the resulting semantic object an *answer set*. Rather than to give the original definition from [LW92], we recall a basic characterization of answer sets of UG-programs that will be of use in the paper. Its proof can be found in [Lif96, MPT99].

Proposition 1. *Let P be a UG-program. A set of atoms M is an answer set to P if and only if M is a stable model of P^+ and a model of P^-. In particular, if M is an answer set to P then M is a model of P.*

Alternating sequences. All approximations to answer sets of UG-programs we study in this paper are defined in terms of alternating sequences and their limits. A sequence (X_i) of sets of literals is *alternating* if

[3] If we treat literals $\mathbf{not}(a)$ as new atoms, P becomes a Horn program and its least model is the least set of literals closed under P.

1. $X_0 \subseteq X_2 \subseteq X_4 \subseteq \ldots$
2. $X_1 \supseteq X_3 \supseteq X_5 \supseteq \ldots$
3. $X_{2i} \subseteq X_{2i+1}$, for every non-negative integer i.

If (X_i) is an alternating sequence, we define $X^l = \bigcup_{i=0}^{\infty} X_{2i}$ and $X^u = \bigcap_{i=0}^{\infty} X_{2i+1}$. We call the pair (X^l, X^u) the *limit* of the alternating sequence (X_i). It follows directly from the definition that for every non-negative integers i and j,

$$X_{2i} \subseteq X^l \subseteq X^u \subseteq X_{2j+1}$$

Alternating sequences are often defined by means of operators that are antimonotone. An operator γ defined on $Lit(U)$ is *antimonotone* if for every two sets $X \subseteq Y \subseteq Lit(U)$, $\gamma(Y) \subseteq \gamma(X)$. Let γ be antimonotone. We define $X_0 = \emptyset$ and $X_{i+1} = \gamma(X_i)$. It is well known (and easy to show) that the sequence (X_i) is alternating. We call (X_i) the *alternating* sequence of γ.

We will consider in the paper the following two operators:

$$\gamma_{P,U}(X) = [P \cup \mathbf{not}(U \setminus X)]^* \cap U \text{ and } \gamma_P(X) = [P \cup \mathbf{not}(U \setminus X)]^*.$$

Both operators are antimonotone and give rise to alternating sequences, say (W_i) and (Y_i). Let (W^l, W^u) and (Y^l, Y^u) be the limits of these sequences, respectively. One can verify that these limits form *alternating pairs*. That is, we have

$$\gamma_{P,U}(W^l) = W^u \text{ and } \gamma_{P,U}(W^u) = W^l \tag{4}$$

and

$$\gamma_P(Y^l) = Y^u \text{ and } \gamma_P(Y^u) = Y^l. \tag{5}$$

One can show that if P is a normal logic program then the alternating sequence of $\gamma_{P,U}$ is precisely the alternating sequence defining the well-founded semantics of P [VRS88, Van93].

One can also show that the limit of the alternating sequence of γ_P is the well-founded model of the normal logic program P' obtained from P by replacing every literal $\mathbf{not}(a)$ with a *new* atom, say a', and adding rules of the form $a' \leftarrow \mathbf{not}(a)$ (the claim holds modulo the correspondence $a' \leftrightarrow \mathbf{not}(a)$). The mapping $P \mapsto P'$ was introduced and studied in [PT95] in the context of revision programs.

Approximating sets of atoms. Let M be a set of atoms. Every pair of sets (T, S) that *approximates* M, that is, such that $T \subseteq M \subseteq S$, implies a lower bound on the complete representation M^c of M:

$$T \cup \{\mathbf{not}(U \setminus S)\} \subseteq M^c.$$

Conversely, every set L of literals such that $L \subseteq M^c$ determines an *approximation* (T, S) of M, where $T = U \cap L$ and $S = \{a \in U \colon \mathbf{not}(a) \notin L\}$. Indeed,

$$U \cap L \subseteq M \subseteq \{a \in U \colon \mathbf{not}(a) \notin L\}.$$

In this way, we establish a bijection between approximations to a set of atoms M and subsets of M^c. It follows that approximations of answer sets can be represented as subsets of their complete representations. We have the following fact.

Proposition 2. *Let P be a UG-program and let T and S be two sets of atoms. For every answer set M of P, if $T \subseteq M \subseteq S$ then $[P \cup T \cup \mathbf{not}(U \setminus S)]^* \subseteq M^c$.*

Proof: We have $T \subseteq M \subseteq S$. Thus, $T \cup \mathbf{not}(U \setminus S) \subseteq M^c$. Let $r = \alpha \leftarrow Body$ be a rule in P such that $Body \subseteq M^c$. It follows that M satisfies the body of r. Since M is an answer set of P, M satisfies α and so, $\alpha \in M^c$. Thus, $T \cup \mathbf{not}(U \setminus S) \subseteq M^c$ and M^c is closed under P. Consequently, $[P \cup T \cup \mathbf{not}(U \setminus S)]^* \subseteq M^c$. □

In the case of normal logic programs, the well-founded model, that is, the limit (W^l, W^u) of the alternating sequence (W_i) of the operator $\gamma_{P,U}$, approximates every stable model (if they exist) and, in some cases determines the existence of a unique stable model.

Theorem 1 ([VRS88, Lif96]). *Let (W^l, W^u) be the well-founded model of a normal logic program P.*

1. *For every stable model M of P, $W^l \cup \mathbf{not}(U \setminus W^u) \subseteq M^c$.*
2. *If $W^l = W^u$, then W^l is a unique stable model for P.*

In the remainder of the paper, we will propose approximations to answer sets of UG-programs generalizing Theorem 1.

3 Approximating Answer Sets Using Operators $\gamma_{P,U}$ and γ_P

Our first approach exploits the fact that every answer set of a UG-program P is a stable model of P^+ (Proposition 1). Let P be a UG-program and let (W^l, W^u) be the limit of the alternating sequence of the operator $\gamma_{P^+,U}$. As we observed, (W^l, W^u) is the well-founded model of P^+. We define

$$Appx_1(P) = [P \cup \mathbf{not}(U \setminus W^u)]^*.$$

By (4), $W^l = [P \cup \mathbf{not}(U \setminus W^u)]^* \cap U$. Hence, $W^l \subseteq Appx_1(P)$ and so, $Appx_1(P)$ contains all literals that are true in the well-founded model (W^l, W^u).

Theorem 2. *Let P be a UG-program. For every answer set M of P, $Appx_1(P) \subseteq M^c$. In addition, if $Appx_1(P)$ is incoherent then P has no answer sets.*

Proof: Let M be an answer set of P. By Proposition 1, M is a stable model of P^+. Let (W^l, W^u) be the well-founded model of P^+. By Theorem 1, $\mathbf{not}(U \setminus W^u) \subseteq M^c$. Moreover, since M is an answer set of P, M is a model of P (Proposition 1, again) and so, M^c is closed under P. Since $Appx_1(P)$ is the least set of literals containing $\mathbf{not}(U \setminus W^u)$ and closed under P, $Appx_1(P) \subseteq M^c$, as claimed. The second part of the assertion follows from the first one. □

We will illustrate this approach with an example.

Example 1. Let us consider the following UG-program P:

$$a \leftarrow \mathbf{not}(b), \mathbf{not}(c) \qquad d \leftarrow \mathbf{not}(b)$$
$$c \leftarrow c, \mathbf{not}(b) \qquad \mathbf{not}(b) \leftarrow$$
$$b \leftarrow \mathbf{not}(d)$$

All but the last rule belong to P^+. The operator $\gamma_{P^+,U}$ determines the following alternating sequence (W_i) of sets:

$$\emptyset \mapsto \{a, b, d\} \mapsto \emptyset \dots .$$

It follows that the well-founded model of P^+ is $(W^l, W^u) = (\emptyset, \{a, b, d\})$. Consequently,

$$Appx_1(P) = [P \cup \{\mathbf{not}(c)\}]^* = \{a, d, \mathbf{not}(b), \mathbf{not}(c)\}.$$

In this case, the well-founded model of P^+ alone provides a weak bound on answer sets of P. The improved bound $Appx_1(P)$, which closes the model under P, provides a much stronger approximation. In fact, only one set M is approximated by $\{a, d, \mathbf{not}(b), \mathbf{not}(c)\}$. This set is $\{a, d\}$ and it happens to be a unique answer set of P.

Let $Q = P \cup \{\mathbf{not}(a) \leftarrow d\}$. Since $Q^+ = P^+$, it follows that $Appx_1(Q) = [Q \cup \{\mathbf{not}(c)\}]^* = \{a, d, \mathbf{not}(a), \mathbf{not}(b), \mathbf{not}(c)\}$. Since $Appx_1(Q)$ is incoherent, Q has no answer sets, a fact that can be verified directly. □

The approximation $Appx_1(P)$, where P is the first program from Example 1, is complete and coherent, and we noted that the unique set of atoms that $Appx_1(P)$ approximates is a unique answer set of P. It is a general property extending Theorem 1(2).

Corollary 1. *Let P be a UG-program. If $Appx_1(P)$ is coherent and complete then $Appx_1(P) \cap U$ is a unique answer set of P.*

Proof: Since $Appx_1(P)$ is coherent and complete, Theorem 2 implies that P has at most one answer set. To prove the assertion it is then enough to show that $M = Appx_1(P) \cap U$ is an answer set of P.

Let (W^l, W^u) be the well-founded model of P^+. Since $Appx_1(P) = [P \cup \mathbf{not}(U \setminus W^u)]^*$, $[P \cup \mathbf{not}(U \setminus W^u)]^*$ is coherent and complete. Consequently,

$$M^c = [P \cup \mathbf{not}(U \setminus W^u)]^*.$$

It follows that $\mathbf{not}(U \setminus W^u) \subseteq \mathbf{not}(U \setminus M)$. Thus, $M^c \subseteq [P \cup \mathbf{not}(U \setminus M)]^*$. It also follows that M^c is closed under the rules in P. Since $\mathbf{not}(U \setminus M) \subseteq M^c$, $[P \cup \mathbf{not}(U \setminus M)]^* \subseteq M^c$. Thus,

$$M^c = [P \cup \mathbf{not}(U \setminus M)]^*.$$

It follows now that M is a model of P^-. Moreover, it also follows that $M = [P^+ \cup \mathbf{not}(U \setminus M)]^*$ and so, M is a stable model of P^+. Thus, M is an answer set of P. □

We will now introduce another approximation to answer sets of a UG-program P. This time, we will use the operator γ_P. Let Y_i be the alternating sequence of the operator γ_P and let (Y^l, Y^u) be the limit of (Y_i). We define

$$Appx_2(P) = Y^l.$$

Theorem 3. *Let P be a UP-program. If M is an answer-set for P then $Appx_2(P) \subseteq M^c$. In addition, if $Appx_2$ is incoherent, then P has no answer sets.*

Proof: Let M be an answer set of P and let (Y_i) be the alternating sequence for the operator γ_P. We will show by induction that for every $i \geq 0$, $Y_{2i} \cap U \subseteq M \subseteq Y_{2i+1}$.

Since $Y_0 = \emptyset$, $Y_0 \cap U \subseteq M$. We will now assume that $Y_{2i} \cap U \subseteq M$ and show that $M \subseteq Y_{2i+1}$. Our assumption implies that $\mathbf{not}(U \setminus M) \subseteq \mathbf{not}(U \setminus Y_{2i})$. Thus, since M is a stable model of P^+, it follows from (3) that

$$M = [P^+ \cup \mathbf{not}(U \setminus M)]^* \cap U \subseteq [P \cup \mathbf{not}(U \setminus M)]^* \subseteq [P \cup \mathbf{not}(U \setminus Y_{2i})]^* = Y_{2i+1}.$$

Next, we assume that $M \subseteq Y_{2i+1}$ and show that $Y_{2i+2} \cap U \subseteq M$. The assumption implies that $\mathbf{not}(U \setminus Y_{2i+1}) \subseteq \mathbf{not}(U \setminus M)$. Thus,

$$\begin{aligned} Y_{2i+2} \cap U &= [P \cup \mathbf{not}(U \setminus Y_{2i+1})]^* \cap U \subseteq [P \cup \mathbf{not}(U \setminus M)]^* \cap U \\ &= [P^+ \cup \mathbf{not}(U \setminus M)]^* \cap U = M. \end{aligned}$$

The last but one equality follows from the fact that M is a model of P^- and the last inequality follows from the fact that M is a stable model of P^+.

From the claim it follows that $M \subseteq Y^u$. Thus, $\mathbf{not}(U \setminus Y^u) \subseteq M^c$. Since M is a model of P, M^c is closed under P. Thus, $Y^l = [P \cup \mathbf{not}(U \setminus Y^u)]^* \subseteq M^c$. □

As before, if the approximation provided by $Appx_2(P)$ is complete and coherent, P has a unique answer set.

Corollary 2. *Let P be a UG-program such that $Appx_2(P)$ is complete and coherent. Then, $Appx_2(P) \cap U$ is a unique answer set of P.*

The following example illustrates our second approach.

Example 2. Let $U = \{a, b\}$. Let P be a UG-program consisting of rules:

$$\begin{aligned} \mathbf{not}(a) &\leftarrow \mathbf{not}(b) \\ b &\leftarrow \mathbf{not}(a) \\ a &\leftarrow \end{aligned}$$

Iterating the operator γ_P results in the following alternating sequence:

$$\emptyset \mapsto \{a, b, \mathbf{not}(a), \mathbf{not}(b)\} \mapsto \{a\} \mapsto \{a, b, \mathbf{not}(a), \mathbf{not}(b)\} \mapsto \ldots .$$

Its limit is $(\{a\}, \{a, b, \mathbf{not}(a), \mathbf{not}(b)\})$ and so, $Appx_2(P) = \{a\}$. □

We conclude this section by showing that the approximations $Appx_1$ and $Appx_2$ are, in general, not comparable.

The following example shows that there is a UG-program P such that $Appx_1(P)$ and $Appx_2(P)$ are coherent and $Appx_2(P)$ is a *proper* subset of $Appx_1(P)$.

Example 3. Let $U = \{a, b, c, d, e\}$ and let P be a UG-program consisting of the rules:

$$\begin{aligned} a &\leftarrow \mathbf{not}(a) & d &\leftarrow \mathbf{not}(c), \mathbf{not}(e) \\ b &\leftarrow \mathbf{not}(a) & e &\leftarrow \\ c &\leftarrow \mathbf{not}(d) & a &\leftarrow c, e \\ & & \mathbf{not}(e) &\leftarrow a, b \end{aligned}$$

Computing $Appx_1(P)$**.** The program P^+ consists of all rules of P except the last one. The alternating sequence of $\gamma_{P^+,U}$ starts as follows:

$$\emptyset \mapsto \{\, a, b, c, d, e \,\} \mapsto \{e\} \mapsto \{\, a, b, c, e \,\} \mapsto \{\, a, c, e \,\} \mapsto \{\, a, c, e \,\} \mapsto \dots\ .$$

Thus, its limit is $(\{a, c, e\}, \{a, c, e\})$ and

$$Appx_1(P) = [P \cup \{a, c, e\} \cup \{\mathbf{not}(b), \mathbf{not}(d)\}]^* = \{a, c, e, \mathbf{not}(b), \mathbf{not}(d)\}.$$

Computing $Appx_2(P)$**.** Iterating the operator γ_P yields the following sequence:

$$\emptyset \mapsto Lit(U) \mapsto \{e\} \mapsto Lit(U) \mapsto \dots\ .$$

Thus, the limit is $(\{e\}, Lit(U))$ and so, $Appx_2(P) = \{e\}$. □

The next example shows that for some programs the opposite is true and the second approximation is strictly more precise.

Example 4. Let $U = \{a, b, c\}$ and let P be a UG-program consisting of the rules:

$$a \leftarrow \mathbf{not}(b) \qquad\qquad c \leftarrow a, b$$
$$b \leftarrow \mathbf{not}(a) \qquad\qquad \mathbf{not}(a) \leftarrow$$

Computing $Appx_1(P)$**.** The alternating sequence of the operator $\gamma_{P^+,U}$ is

$$\emptyset \mapsto \{a, b, c\} \mapsto \emptyset \mapsto \dots\ .$$

Thus,

$$Appx_1(P) = P^* = \{\mathbf{not}(a), b\}.$$

Computing $Appx_2(P)$**.** Iterating γ_P yields:

$$\emptyset \mapsto Lit(U) \mapsto \{\mathbf{not}(a), b\} \mapsto \{\mathbf{not}(a), b, \mathbf{not}(c)\} \mapsto \{\mathbf{not}(a), b, \mathbf{not}(c)\} \mapsto \dots\ .$$

Thus, $Appx_2(P) = \{\mathbf{not}(a), b, \mathbf{not}(c)\}$. □

4 Strong Approximation

Let P be a UG-program and $Z \subseteq Lit(U)$ a set of literals (not necessarily *coherent*). By the *weak reduct* of P with respect to Z we mean the program P^Z_w obtained from P by:

1. removing all rules that contain in the body a literal $\mathbf{not}(a)$ such that $a \in Z$ and $\mathbf{not}(a) \notin Z$;
2. removing from the bodies of the remaining rules all literals $\mathbf{not}(a)$ such that $a \notin Z$.

Let us note that if $a \in Z$ and $\mathbf{not}(a) \in Z$, $\mathbf{not}(a)$ will not be removed from the rules that remain after Step 1.

Let Z be a set of literals, $Z \subseteq Lit(U)$. We define

$$\gamma^w_P(Z) = [P^Z_w]^*.$$

In general, the operator γ^w_P is not antimonotone. Thus, the sequence (Z_i) obtained by iterating γ^w_P (starting with the empty set) in general is not alternating.

Example 5. Let P be a UG-program consisting of the rules:

$$\begin{array}{rcr} a \leftarrow \mathbf{not}(b) & & c \leftarrow \mathbf{not}(d) \\ b \leftarrow & & d \leftarrow \\ \mathbf{not}(b) \leftarrow \mathbf{not}(c) & & \end{array}$$

By the definition, $Z_0 = \emptyset$. When computing P^{Z_0}, no rule is removed in Step 1 of the definition of the weak reduct, and every literal of the form $\mathbf{not}(a)$ is removed from the bodies of rules in P. Thus, $Z_1 = \{a, b, c, d, \mathbf{not}(b)\}$. When computing $P_w^{Z_1}$, we observe that $\mathbf{not}(b) \in Z_1$. Thus, the first rule is not removed despite the fact that $b \in Z_1$. Hence, we have:

$$P_w^{Z_1} = \left\{ \begin{array}{l} a \leftarrow \mathbf{not}(b) \\ b \leftarrow \\ d \leftarrow \end{array} \right\}, \quad \text{and so,} \;\; Z_2 = \{b, d\}.$$

In the next step, we compute:

$$P_w^{Z_2} = \left\{ \begin{array}{r} b \leftarrow \\ \mathbf{not}(b) \leftarrow \\ d \leftarrow \end{array} \right\}, \quad \text{and so,} \;\; Z_3 = \{b, d, \mathbf{not}(b)\}.$$

When computing $P_w^{Z_3}$, the rule $a \leftarrow \mathbf{not}(b)$ is again *not* removed in Step 1. Thus,

$$P_w^{Z_3} = \left\{ \begin{array}{r} a \leftarrow \mathbf{not}(b) \\ b \leftarrow \\ \mathbf{not}(b) \leftarrow \\ d \leftarrow \end{array} \right\}, \quad \text{and so,} \;\; Z_4 = \{a, b, d, \mathbf{not}(b)\}.$$

We note that Z_4 is *not* a subset of Z_3. Thus, for this program P, the sequence (Z_i) is not alternating. □

In the remainder of this section we show that under some conditions the sequence (Z_i) is alternating and may be used to approximate answer sets of UG-programs. We first establish a lemma providing conditions, under which $[P_w^X]^*$ is antimonotone in X.

Lemma 1. *Let P be a UG-program, X and X' be sets of literals such that $X \subseteq X'$. Moreover, let at least one of the following conditions hold:*

1. *X' is coherent*
2. *$X \subseteq [P_w^{X'}]^*$ and $[P_w^{X'}]^*$ is coherent.*
3. *$[P_w^{X'}]^* \subseteq X$*
4. *$X \subseteq [P_w^X]^*$ and $[P_w^X]^*$ is coherent.*

Then $[P_w^{X'}]^* \subseteq [P_w^X]^*$.

The next lemma describes two properties of $[P_w^X]^*$ under the assumption that X is coherent.

Lemma 2. *Let P be a UG-program and X a coherent set of literals, $X \subseteq Lit(U)$.*

1. $[P_w^X]^* = [P_w^{X \cap U}]^*$.
2. $[P_w^X]^* = [(P^+)_w^{X \cap U}]^* \cup \mathbf{not}(X') = [(P^+)^{X \cap U}]^* \cup \mathbf{not}(X')$, *where X' is the set of atoms such that $a \in X'$ if and only if there is a rule $\mathbf{not}(a) \leftarrow Body$ in $(P^-)_w^X$ such that $[(P^+)_w^{X \cap U}]^* \models Body$.*

We can now prove the following characterization of answer sets of UG-programs.

Lemma 3. *Let P be a UG-program, $M \subseteq U$ a set of atoms, and N a set of atoms consisting of all atoms $a \in U$ such that $a \notin M$ and there is a rule $\mathbf{not}(a) \leftarrow Body$ in P such that $M \models Body$. Then M is an answer set of P if and only if $[P_w^M]^* = M \cup \mathbf{not}(N)$.*

Proof: ($\Rightarrow$) By Proposition 1, M is a stable model of P^+ and a model of P^-. In particular, $[(P^+)^M]^* = M$. Let X' be the set specified in Lemma 2(2), defined for $X = M$. Since $[(P^+)^M]^* = M$ and M is a model of P^-, for every $a \in X'$, $a \notin M$. Thus, $X' = N$ and the assertion follows from Lemma 2(2).

($\Leftarrow$) It follows from Lemma 2(2) that $M = [(P^+)^M]^*$. Thus, M is stable model of P^+. Let us consider a rule $\mathbf{not}(a) \leftarrow Body$ from P^- such that M satisfies $Body$. Let $Body'$ consist of all atoms in $Body$. It follows that $\mathbf{not}(a) \leftarrow Body'$ is a rule in $(P^-)_w^M$. Since $M \models Body$, $M \models Body'$. Thus, by Lemma 2(2), $\mathbf{not}(a) \in [P_w^M]^*$. Since $[P_w^M]^* = M \cup \mathbf{not}(N)$, $a \in \mathbf{not}(N)$ which, in turn, implies $a \notin M$. It follows that M is a model of P^- and so, an answer set of P. □

The results we presented above allow us to prove that as long as the lower (even) terms of the sequence (Z_i) are coherent, the sequence behaves as an alternating one.

Proposition 3. *Let i be an integer, $i \geq 0$, such that Z_{2i} is coherent. Then*

1. $Z_0 \subseteq Z_2 \subseteq \ldots \subseteq Z_{2i}$
2. $Z_1 \supseteq Z_3 \supseteq \ldots \supseteq Z_{2i+1}$
3. $Z_{2i} \subseteq Z_{2i+1}$.

This last proposition is crucial for the definition of our third approximation. Let us consider the sequence (Z_i). If for every i, Z_{2i} is coherent, Proposition 3 implies that the sequence (Z_i) is alternating. Let (Z^l, Z^u) be the limit of (Z_i). We define

$$Appx_3(P) = Z^l \cup \{\mathbf{not}(a) \colon a \in U \setminus Z^u\}.$$

Otherwise, there is i such that Z_{2i} is incoherent. In this case, we say that $Appx_3(P)$ is undefined.

Theorem 4. *Let P be a UG-program. If M is an answer set of P then $Appx_3(P)$ is defined and $Appx_3(P) \subseteq M^c$. If $Appx_3(P)$ is not defined, then P has no answer sets.*

Proof: The second part of the assertion follows from the first one. To prove the first part of the assertion, we will show that for every $i \geq 0$, $Z_{2i} \subseteq M^c$, and $M \subseteq Z_{2i+1}$.

We proceed by induction on i. If $i = 0$, then $Z_0 = \emptyset \subseteq M^c$. We now assume that $Z_{2i} \subseteq M^c$ and prove that $M \subseteq Z_{2i+1}$.

Since $Z_{2i} \subseteq M^c$ and M^c is coherent, Z_{2i} is coherent, too. By Lemma 1 (applied to $X = Z_{2i}$ and $X' = M^c$, under the assumption (4)), $[P_w^{M^c}]^* \subseteq [P_w^{Z_{2i}}]^*$. Thus, $[P_w^{M^c}]^* \subseteq Z_{2i+1}$. By Lemma 2(1), $[P_w^M]^* \subseteq Z_{2i+1}$. By Lemma 3, $M \subseteq [P_w^M]^*$. Therefore, $M \subseteq Z_{2i+1}$.

Next, we assume that $M \subseteq Z_{2i+1}$ and prove that $Z_{2i+2} \subseteq M^c$. Let us note that $Z_{2i+2} = [P_w^{Z_{2i+1}}]^*$ and that by Lemma 3, $[P_w^M]^* \subseteq M^c$. Thus, it will suffice to show that $[P_w^{Z_{2i+1}}]^* \subseteq [P_w^M]^*$. To this end, we note that by Lemma 3, $M \subseteq [P_w^M]^*$ and so Lemma 1 applies (under the condition (4)) to $X = M$ and $X' = Z_{2i+1}$, and implies the required inclusion.

It follows that $Z^l \subseteq M^c$ and that $M \subseteq Z^u$. If $a \notin Z^u$, then $a \notin M$ and so, $\mathbf{not}(a) \in M^c$. Thus, $Appx_3(P) = Z^l \cup \mathbf{not}(U \setminus Z^u) \subseteq M^c$. □

Example 6. Let P be a UG-program consisting of the rules:

$$\begin{array}{rlcrl} \mathbf{not}(a) & \leftarrow & \quad & \mathbf{not}(d) & \leftarrow \mathbf{not}(c) \\ a & \leftarrow \mathbf{not}(b) & & d & \leftarrow \mathbf{not}(e) \\ b & \leftarrow \mathbf{not}(a) & & e & \leftarrow \mathbf{not}(d) \\ c & \leftarrow a, b & & f & \leftarrow d, e \end{array}$$

Iterating the operator γ_P^w results in the following sequence:

$$\emptyset \mapsto \{a, b, c, d, e, f, \mathbf{not}(a), \mathbf{not}(d)\} \mapsto \{\mathbf{not}(a), b\} \mapsto \{b, d, e, f, \mathbf{not}(a), \mathbf{not}(d)\}$$
$$\mapsto \{b, e, \mathbf{not}(a), \mathbf{not}(d)\} \mapsto \{b, e, \mathbf{not}(a), \mathbf{not}(d)\} \mapsto \ldots .$$

Thus, the sequence (Z_i) is alternating. Its limit is (Z^l, Z^u), where $Z^l = Z^u = \{b, e, \mathbf{not}(a), \mathbf{not}(d)\}$. Thus,

$$Appx_3(P) = Z^l \cup \mathbf{not}(U \setminus Z^u) = \{b, e, \mathbf{not}(a), \mathbf{not}(c), \mathbf{not}(d), \mathbf{not}(f)\}.$$

Since $Appx_3(P)$ is coherent and complete, P has a unique answer set, $\{b, e\}$. This example also demonstrates that Z^u can improve on the bound provided by Z^l itself. □

5 Properties of $Appx_3$

In this section we will show that if $Appx_3$ is defined then it is stronger than the other two approximations. We recall that if $Appx_3(P)$ is undefined, then P has no answer sets, that is, P is *inconsistent*. It follows that for all *consistent* UG-programs, $Appx_3$ is stronger than the the other two approximations.

Theorem 5. *LetP be a UG-program. If $Appx_3(P)$ is defined then*

$$Appx_1(P) \cup Appx_2(P) \subseteq Appx_3(P)$$

There are programs which show that$Appx_3$ is strictly stronger.

Example 7. Let P be the UG-program from Example 4. We recall that $Appx_1(P) = \{\mathbf{not}(a), b\}$. Let us compute $Appx_3(P)$. By iterating the operator γ_w^P, we obtain the following sequence:

$$Z_0 = \emptyset \mapsto Z_1 = \{a, b, c, \mathbf{not}(a)\} \mapsto Z_2 = \{\mathbf{not}(a), b\} \mapsto Z_3 = \{\mathbf{not}(a), b\} \ldots.$$

Hence, $Appx_3(P) = \{\mathbf{not}(a), b, \mathbf{not}(c)\}$ and $Appx_1(P)$ is a *proper* subset of $Appx_3(P)$. □

Example 8. Let P be the UG-program from Example 3. We recall that $Appx_2(P) = \{e\}$. To compute $Appx_3(P)$, we note that by iterating the operator γ_w^P we get the following sequence:

$$Z_0 = \emptyset \mapsto Z_1 = \{a, b, c, d, e, \mathbf{not}(e)\} \mapsto Z_2 = \{e\} \mapsto$$

$$Z_3 = \{a, b, c, e, \mathbf{not}(e)\} \mapsto Z_4 = \{a, c, e\} \mapsto Z_5 = \{a, c, e\} \ldots.$$

Hence, $Appx_3(P) = \{a, \mathbf{not}(b), c, \mathbf{not}(d), e\}$ and $Appx_2(P)$ is a *proper* subset of $Appx_3(P)$. □

Finally, we show that if $Appx_3(P)$ is defined and complete then P has a unique answer set.

Corollary 3. *Let P be a UG-program such that $Appx_3(P)$ is defined and complete. Then $Appx_3(P) \cap U$ is an answer set of P and P has no other answer sets.*

6 Corollaries for the Case of Revision Programs

Revision programming [MT98] is a formalism for describing and enforcing constraints on databases. The main concepts in the formalism are an initial database, a revision program, and justified revisions.

Expressions of the form **in**(a) and **out**(a) ($a \in U$) are *revision literals*. Intuitively, **in**(a) (respectively, **out**(a)) means that atom a is in (respectively, is not in) a database.

A *revision program* consists of rules $\alpha \leftarrow \alpha_1, \ldots, \alpha_n$, where α, $\alpha_i, \ldots, \alpha_n$ are revision literals. Given a revision program P and an initial database I, [MT98] defined *P-justified revisions* of I to represent revisions that satisfy the constraints of P, are "grounded" in P and I, and differ minimally from the initial database.

As we mentioned earlier, unitary general programs are equivalent to revision programs. The equivalence is established by the so called *shifting theorem* [MPT99], which allows us to reduce any pair (P, I), where P is a revision program and I is an initial database, to a unitary general program so that P-justified revisions of I correspond to answer sets of the unitary general program. Consequently, all results of our paper imply results about approximations of justified revisions. Formal descriptions of $Appx_1$, $Appx_2$, and $Appx_3$ for revision programs can be found in [Piv05]. Approximations $Appx_1$ and $Appx_2$ for revision programs were originally described in [Piv01].

Acknowledgments. Inna Pivkina was supported by the NSF-funded ADVANCE Institutional Transformation Program at New Mexico State University, Grant No. 0123690, and NMSU College of Arts and Sciences Research Center Grant No. 01-3-43891. The other two authors were supported by the NSF Grants No. 0097278 and 0325063.

References

[BTK93] A. Bondarenko, F. Toni, and R.A. Kowalski. An assumption-based framework for non-monotonic reasoning. In A. Nerode and L. Pereira, editors, *Logic programming and non-monotonic reasoning (Lisbon, 1993)*, pages 171–189, Cambridge, MA, 1993. MIT Press.

[GL88] M. Gelfond and V. Lifschitz. The stable semantics for logic programs. In *Proceedings of the 5th International Conference on Logic Programming*, pages 1070–1080. MIT Press, 1988.

[GL91] M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991.

[Lif96] V. Lifschitz. Foundations of logic programming. In *Principles of Knowledge Representation*, pages 69–127. CSLI Publications, 1996.

[LW92] V. Lifschitz and T.Y.C. Woo. Answer sets in general nonmonotonic reasoning. In *Proceedings of the 3rd international conference on principles of knowledge representation and reasoning, KR '92*, pages 603–614, San Mateo, CA, 1992. Morgan Kaufmann.

[MPT99] V. W. Marek, I. Pivkina, and M. Truszczyński. Revision programming = logic programming + integrity constraints. In G. Gottlob, E. Grandjean, and K. Seyr, editors, *Computer Science Logic, 12th International Workshop, CSL'98*, volume 1584 of *Lecture Notes in Computer Science*, pages 73–89. Springer, 1999.

[MPT02] V. W. Marek, I. Pivkina, and M. Truszczyński. Annotated revision programs. *Artificial Intelligence Journal*, 138:149–180, 2002.

[MT98] W. Marek and M. Truszczyński. Revision programming. *Theoretical Computer Science*, 190(2):241–277, 1998.

[Piv01] I. Pivkina. Revision programming: a knowledge representation formalism. PhD dissertation, University of Kentucky, 2001.

[Piv05] I. Pivkina. Defining well-founded semantics for revision programming Technical Report NMSU-CS-2005-001, New Mexico State University, Computer Science Department, 2005.

[PT95] T.C. Przymusiński and H. Turner. Update by means of inference rules. In *Logic programming and nonmonotonic reasoning (Lexington, KY, 1995)*, volume 928 of *Lecture Notes in Computer Science*, pages 156–174, Berlin, 1995. Springer.

[SNS02] P. Simons, I. Niemelä, and T. Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138:181–234, 2002.

[SNV95] V.S. Subrahmanian, D. Nau, and C. Vago. WFS + branch bound = stable models. *IEEE Transactions on Knowledge and Data Engineering*, 7:362–377, 1995.

[Van93] A. Van Gelder. The alternating fixpoint of logic programs with negation. *Journal of Computer and System Sciences*, 47(1):185–221, 1993.

[VRS88] A. Van Gelder, K.A. Ross, and J.S. Schlipf. Unfounded sets and well-founded semantics for general logic programs. In *ACM Symposium on Principles of Database Systems*, pages 221–230, 1988.

On Modular Translations and Strong Equivalence

Paolo Ferraris

Department of Computer Sciences, The University of Texas at Austin,
Austin TX 78712, USA
otto@cs.utexas.edu

Abstract. Given two classes of logic programs, we may be interested in modular translations from one class into the other that are sound with respect to the answer set semantics. The main theorem of this paper characterizes the existence of such a translation in terms of strong equivalence. The theorem is used to study the expressiveness of several classes of programs, including the comparison of cardinality constraints with monotone cardinality atoms.

1 Introduction

The notion of an answer set (or "stable model"), originally defined in [Gelfond and Lifschitz, 1988], was extended to more general logic programs in various ways. In Fig. 1 we see some examples of extensions of the class of "traditional" rules studied in that paper, and also some subclasses of that class. The language in each line of the table contains the languages shown in the previous lines.

When we compare the expressiveness of two classes of rules R and R', several criteria can be used. First, we can ask whether for any R-program (that is, a set of rules of the type R) one can find an R'-program that has exactly the same answer sets. (That means, in particular, that the R'-program does not use "auxiliary atoms" not occurring in the given R-program.) From this point of view, the classes of rules shown in Fig. 1 can be divided into three groups: a UR- or PR-program has a unique answer set; TR-, TRC- and DR-programs may have many answer sets, but its answer sets always have the "anti-chain" property (one cannot be a proper subset of another); a NDR- or RNE-program can have an arbitrary collection of sets of atoms as its collection of answer sets.

Another comparison criterion is based on the computational complexity of the problem of the existence of an answer set. We pass, in the complexity hierarchy, from P in case of UR- and PR-programs, to NP in case of TR- and TRC-programs [Marek and Truszczyński, 1991], and finally to Σ_2^P for more complex kinds of programs [Eiter and Gottlob, 1993].

A third criterion consists in checking whether every rule in R is strongly equivalent [Lifschitz *et al.*, 2001] to an R'-program. From this point of view, PR is essentially more expressive than UR: we will see at the end of Sect. 3 that

C. Baral et al. (Eds.): LPNMR 2005, LNAI 3662, pp. 79–91, 2005.
© Springer-Verlag Berlin Heidelberg 2005

class of rules	syntactic form
UR	unary rules: $a \leftarrow$ (also written as simply a) and $a \leftarrow b$
PR	positive rules: $a \leftarrow b_1, \dots, b_n$
TR	traditional rules: $a \leftarrow b_1, \dots, b_n, not\ c_1, \dots, not\ c_m$
TRC	TRs + constraints: TRs and $\leftarrow b_1, \dots, b_n, not\ c_1, \dots, not\ c_m$
DR	disjunctive rules: $a_1; \dots; a_p \leftarrow b_1, \dots, b_n, not\ c_1, \dots, not\ c_m$
NDR	negational disjunctive rules: $a_1, \dots; a_p; not\ d_1; \dots; not\ d_q \leftarrow b_1, \dots, b_n, not\ c_1, \dots, not\ c_m$
RNE	rules with nested expressions: $F \leftarrow G$

Fig. 1. A classification of logic programs under the answer set semantics. Here a, b, c, d stand for propositional atoms. F, G stand for nested expressions without classical negation [Lifschitz *et al.*, 1999], that is, expressions formed from atoms, $\top$ and $\bot$, using conjunction (,), disjunction (;) and negation as failure (*not*).

$a \leftarrow b, c$ is not strongly equivalent to any set of unary rules. Furthermore, TRC and DR are essentially different from each other, since no program in TRC is strongly equivalent to the rule $p; q$ in DR [Turner, 2003, Proposition 1].

A fourth comparison criterion is based on the existence of a translation from R-programs to R'-programs that is not only sound (that is, preserves the program's answer sets) but is also modular: it can be applied to a program rule-by-rule. For instance, [Janhunen, 2000] showed that there is no modular translation from PR to UR and from TR to PR [1]. On the other hand, RNE can be translated into NDR by a modular procedure similar to converting formulas to conjunctive normal form [Lifschitz *et al.*, 1999].

The main theorem of this paper shows that under some general conditions, the last two criteria — the one based on strong equivalence and the existence of a sound modular translation — are equivalent to each other. This offers a method to prove that there is no modular translation from R to R' by finding a rule in R that is not strongly equivalent to any R'-program. For instance, in view of the Proposition 1 from [Turner, 2003] mentioned above, no modular translation exists from DR to TRC.

To apply the main theorem to other cases, we need to learn more about the strong equivalence relations between a single rule of a language and a set of rules. We show that for many rules r in NDR, any NDR-program that is strongly equivalent to r contains a rule that is at least as "complex" as r. This fact will allow us to conclude that all classes UR, PR, TR, TRC, DR and NDR are essentially different from each other in terms of strong equivalence. In view

[1] His results are actually stronger, see Sect. 7 below.

of the main theorem, it follows that they are essentially different from each other in the sense of the modular translation criterion as well.

Finally, we show how to apply our main theorem to programs with weight constraints [Simons *et al.*, 2002]. As a result, we find that it is not possible to translate programs with weight constraints into programs with monotone cardinality atoms [Marek and Niemelä, 2004] in a modular way (unless the translation introduces auxiliary atoms).

The paper continues with the statement of our main theorem (Sect. 2). In Sect. 3, we study the expressiveness of subclasses of NDR in terms of strong equivalence and modular translations. We move to the study of cardinality constraints in Sect. 4. Section 5 provides some background needed for the proof of some of the claims of this paper (Sect. 6).

2 Modular Transformations and Strong Equivalence

We assume that the reader is familiar with the concept of an answer set for the classes of logic programs in Fig. 1 (the semantics for the class RNE, which is applicable to all its subclasses, is reproduced in Sect. 5.1). A *program* is a subset of RNE. Two programs Π_1 and Π_2 are *strongly equivalent* if, for every program Π, $\Pi_1 \cup \Pi$ and $\Pi_2 \cup \Pi$ have the same answer sets. A *(modular) transformation* is a function f such that

- $\mathrm{Dom}(f) \subseteq$ RNE, and
- for every rule $r \in \mathrm{Dom}(f)$, $f(r)$ is a program such that every atom occurring in it occurs in r also.

A transformation f is *sound* if, for every program $\Pi \subseteq \mathrm{Dom}(f)$, Π and $\bigcup_{r\in\Pi} f(r)$ have the same answer sets.

For example, the transformation defined in the proof of Proposition 7 from [Lifschitz *et al.*, 1999], which eliminates nesting from a program with nested expressions, is a sound transformation. For instance, for this transformation f,

$$f(a \leftarrow b; c) = \{a \leftarrow b,\ a \leftarrow c\}.$$

As another example of a sound transformation, consider the transformation f with $\mathrm{Dom}(f) = NDR$, where

$$f(not\ d_1; \ldots; not\ d_q \leftarrow b_1, \ldots, b_n, not\ c_1, \ldots, not\ c_m) = \\ \{\leftarrow b_1, \ldots, b_n, d_1, \ldots, d_q, not\ c_1, \ldots, not\ c_m\}$$

and $f(r) = \{r\}$ for the other rules r in NDR. On the other hand, the familiar method of eliminating constraints from a program that turns $\leftarrow p$ into $q \leftarrow p, not\ q$ is not a transformation in the sense of our definition, because it introduces an atom q that doesn't occur in $\leftarrow p$.

This is the theorem that relates strong equivalence and modular transfomations:

Theorem 1 (Main Theorem). *For every transformation f such that* $\mathrm{Dom}(f)$ *contains all unary rules, f is sound iff, for each $r \in \mathrm{Dom}(f)$, $f(r)$ is strongly equivalent to r.*

Our definition of transformation requires that all atoms that occur in $f(r)$ occur in r also. The following counterexample shows that without this assumption the assertion of the Main Theorem would be incorrect. Let p and q be two atoms, and, for each rule $r = F \leftarrow G$ in RNE, let $f_1(r)$ be

$$
\begin{array}{l}
F \leftarrow G, \mathit{not}\ p \\
F \leftarrow G, \mathit{not}\ q \\
F_{p \leftrightarrow q} \leftarrow G_{p \leftrightarrow q}, \mathit{not\ not}\ p, \mathit{not\ not}\ q.
\end{array}
$$

where $F_{p \leftrightarrow q}$ and $G_{p \leftrightarrow q}$ stand for F and G with all the occurrences of p replaced by q and vice versa. Note that f_1 is not a transformation as defined in this paper since q occurs in $f_1(p \leftarrow \top)$. It can also be shown that $p \leftarrow \top$ and $f_1(p \leftarrow \top)$ are not strongly equivalent. However, the "transformation" is sound:

Proposition 1. *For any program Π, Π and $\bigcup_{r \in \Pi} f_1(r)$ have the same answer sets.*

Without the assumption that $\mathrm{UR} \subseteq \mathrm{Dom}(f)$, the Main Theorem would not be correct either. We define a transformation f_2 such that $\mathrm{Dom}(f_2)$ consists of all rules of DR where all atoms in the body are prefixed by negation as failure, and the head is nonempty: the rules have the form

$$
a_1; \ldots; a_p \leftarrow \mathit{not}\ c_1, \ldots, \mathit{not}\ c_m. \tag{1}
$$

with $p > 0$. For each rule r of the form (1), $f_2(r)$ is defined as

$$
\{a_i \leftarrow \mathit{not}\ a_1, \ldots, \mathit{not}\ a_{i-1}, \mathit{not}\ a_{i+1}, \ldots, \mathit{not}\ a_p, \mathit{not}\ c_1, \ldots, \mathit{not}\ c_m \ : \ 1 \leq i \leq p\}.
$$

It is easy to see that $f_2(p; q)$ is not strongly equivalent to $p; q$. However, this transformation is sound:

Proposition 2. *For any program $\Pi \subseteq \mathrm{Dom}(f_2)$, Π and $\bigcup_{r \in \Pi} f_2(r)$ have the same answer sets.*

3 Applications: Negational Disjunctive Rules

In order to apply the Main Theorem to modular translations, we first need to study some properties of strong equivalence. We focus on the class NDR.

If r is

$$
a_1, \ldots; a_p; \mathit{not}\ d_1; \ldots; \mathit{not}\ d_q \leftarrow b_1, \ldots, b_n, \mathit{not}\ c_1, \ldots, \mathit{not}\ c_m
$$

define

$$head^+(r) = \{a_1, \ldots, a_p\} \qquad head^-(r) = \{d_1, \ldots, d_q\}$$
$$body^+(r) = \{b_1, \ldots, b_n\} \qquad body^-(r) = \{c_1, \ldots, c_m\}.$$

We say that r is *basic* if every pair of these sets, except possibly for the pair $head^+(r)$, $head^-(r)$, is disjoint.

Any nonbasic rule can be easily simplified: it is either strongly equivalent to the empty program or contains redundant terms in the head. A basic rule, on the other hand, cannot be simplified if it contains at least one nonnegated atom in the head:

Theorem 2. *Let r be a basic rule in NDR such that $head^+(r) \neq \emptyset$. Every program subset of NDR that is strongly equivalent to r contains a rule r' such that*

$$head^+(r) \subseteq head^+(r') \qquad body^+(r) \subseteq body^+(r')$$
$$head^-(r) \subseteq head^-(r') \qquad body^-(r) \subseteq body^-(r')$$

This theorem shows us that for most basic rules r (the ones with at least one positive element in the head), every program strongly equivalent to r must contain a rule that is at least as "complex" as r.

Given two subsets R and R' of RNE, a *(modular) translation* from R to R' is a transformation f such that $\mathrm{Dom}(f) = R$ and $f(r)$ is a subset of R' for each $r \in \mathrm{Dom}(f)$. Using Theorems 1 and 2, we can differentiate between the classes of rules in Fig. 1 in terms of modular translations:

Proposition 3. *For any two languages R and R' among UR, PR, TRC, TR, DR and NDR such that $R' \subset R$, there is no sound translation from R to R'.*

Theorems 1 and 2 allow us also to differentiate between subclasses of NDR described in terms of the sizes of various parts of the rule. Define, for instance PBR_i ("positive body of size i") as the set of rules r of NDR such that $|body^+(r)| \leq i$. We can show that, for every $i \geq 0$, there is no sound translation from PBR_{i+1} to PBR_i (or even from $\mathrm{PBR}_{i+1} \cap \mathrm{PR}$ to PBR_i). Similar properties can be stated in terms of the sizes of $body^-(r)$, $head^+(r)$ and $head^-(r)$.

Another consequence of Theorem 2 is in terms of (absolute) tightness of a program [Erdem and Lifschitz, 2003, Lee, 2005]. Tightness is an important property of logic programs: if a program is tight then its answer sets can be equivalently characterized by the satisfaction of a set of propositional formulas of about the same size. Modular translations usually don't make nontight programs tight:

Proposition 4. *Let R be any subset of NDR that contains all unary rules, and let f be any sound translation from R to NDR. For every nontight program $\Pi \subset R$ consisting of basic rules only, $\bigcup_{r \in \Pi} f(r)$ is nontight.*

4 Applications: Programs with Cardinality Constraints

4.1 Syntax

We briefly review the syntax of programs with cardinality constraints [Simons *et al.*, 2002].

A *rule element* is an atom possibly prefixed with negation as failure symbol *not*. A *cardinality constraint* is an expression of the form

$$L\{c_1, \ldots, c_m\}U \tag{2}$$

where

- each of L, U is (a symbol for) an integer or $-\infty$, $+\infty$,
- $m \geq 0$, and
- $c_1, \ldots, c_m$ are rule elements.

As an abbreviation, L can be omitted if $L = -\infty$; similarly, we can drop U if $U = +\infty$. A *rule with cardinality constraints* is an expression of the form

$$C_0 \leftarrow C_1, \ldots, C_n \tag{3}$$

where $C_0, \ldots, C_n$ $(n \geq 0)$ are cardinality constraints. A *program with cardinality constraints* is a set of rules with cardinality constraints.

Let CCR denote the set of all rules with cardinality constraints. A straightforward generalization of the definition of a transformation allows us to talk about sound translations between subclasses of CCR, and also between a class of CCR and a subclass of RNE. The concept of (modular) transformations and translations can be extended to programs with cardinality constraints: we can have translations between subclasses of CCR, and from/to subclasses of RNE. The definition of the soundness for those translations follows as well.

Another class of programs similar to the one with cardinality constraints — programs with monotone cardinality atoms — has been defined in [Marek and Niemelä, 2004]. The results of that paper show that rules with monotone cardinality atoms are essentially identical to rules with cardinality constraints that don't contain negation as failure; we will denote the set of all such rules by PCCR ("positive cardinality constraints").

4.2 Translations

First of all, we show how programs with cardinality constraints are related to the class NDR. Let SNDR (Simple NDR) be the language consisting of rules of the form

$$a; not\ d_1; \ldots; not\ d_q \leftarrow b_1, \ldots, b_n, not\ c_1, \ldots, not\ c_m \tag{4}$$

and

$$\leftarrow b_1, \ldots, b_n, not\ c_1, \ldots, not\ c_m. \tag{5}$$

Proposition 5. *There exist sound translations from SNDR to CCR, and back.*

If we don't allow negation in cardinality constraints, another relationship holds. We define the class VSNDR (Very Simple NDR) consisting of rules of the form

$$a; not\ a \leftarrow b_1, \ldots, b_n, not\ c_1, \ldots, not\ c_m \tag{6}$$

and of the form (5).

Proposition 6. *There exist sound translations from VSNDR to PCCR, and back.*

Using Theorems 1 and 2, we can prove:

Proposition 7. *There is no sound translation from CCR to PCCR.*

Since the class PCCR is essentially identical to the class of rules with monotone cardinality atoms, we have that programs with cardinality constraints are essentially more expressive than programs with monotone cardinality atoms.

5 Background for Proofs

5.1 Answer Set Semantics for RNE

The semantics of programs is characterized by defining when a set X of atoms is an answer set for a program Π. As a preliminary step, we define when a set X of atoms *satisfies* a formula F (symbolically, $X \models F$), as follows:

- for an atom a, $X \models a$ if $a \in X$
- $X \models \top$
- $X \not\models \bot$
- $X \models (F, G)$ if $X \models F$ and $X \models G$
- $X \models (F; G)$ if $X \models F$ or $X \models G$
- $X \models not\ F$ if $X \not\models F$.

We say that X *satisfies* a program Π (symbolically, $X \models \Pi$) if, for every rule $F \leftarrow G$ in Π, $X \models F$ whenever $X \models G$.

The *reduct* Π^X of a program Π with respect to a set X of atoms is obtained by replacing each outermost formula of the form *not* F (that is, every formula of the form *not* F not in the scope of negation as failure) by $\bot$, if $X \models F$, and by $\top$ otherwise.

The concept of an answer set is defined first for programs not containing negation as failure: a set X of atoms is an *answer set* for such a program Π if X is a minimal set satisfying Π. For an arbitrary program Π, we say that X is an *answer set* for Π if X is an answer set for the reduct Π^X.

5.2 Strong Equivalence

The following lemma is the main criterion that we use to check strong equivalence in most of the proofs. It can be proved in a way similar to the equivalence criterion from [Turner, 2003].

Lemma 1. *Let A be the set of atoms occurring in programs Π_1 and Π_2. Π_1 and Π_2 are strongly equivalent iff, for each $Y \subseteq A$,*

- *$Y \models \Pi_1^Y$ iff $Y \models \Pi_2^Y$, and*
- *if $Y \models \Pi_1^Y$ then, for each $X \subset Y$, $X \models \Pi_1^Y$ iff $X \models \Pi_2^Y$.*

Next lemma can be easily proven under the characterization of strong equivalence as stated in [Lifschitz *et al.*, 2001].

Lemma 2. *Let P_1 and P_2 be sets of programs. If each program in P_1 is strongly equivalent to a program in P_2 and vice versa, then $\bigcup_{\Pi \in P_1} \Pi$ and $\bigcup_{\Pi \in P_2} \Pi$ are strongly equivalent.*

Finally, we will use another property of strong equivalence from [Lifschitz *et al.*, 2001]:

Lemma 3. *Two programs P_1 and P_2 are strongly equivalent iff, for every program $\Pi \subseteq UR$, $\Pi_1 \cup \Pi$ and $\Pi_2 \cup \Pi$ have the same answer sets.*

6 Proofs

6.1 Proof of the Main Theorem

Main Theorem. *For every transformation f such that* $\mathrm{Dom}(f)$ *contains all unary rules, f is sound iff, for each $r \in R$, $f(r)$ is strongly equivalent to r.*

The proof from right to left is a direct consequence of Lemma 2: if $f(r)$ is strongly equivalent to r for every rule $r \in R$, then for any $\Pi \subseteq R$, Π and $\bigcup_{r \in \Pi} f(r)$ are strongly equivalent, and consequently have the same answer sets.

In the proof from left to right, we first consider the case when r is a unary rule, and then extend the conclusion to arbitrary rules. In the rest of this section, f is an arbitrary sound transformation such that $\mathrm{Dom}(f)$ contains all unary rules. By a and b we denote distinct atoms.

Lemma 4. *For every fact a, $\{a\}$ and $f(a)$ are strongly equivalent.*

Proof. For every program Π that has $\{a\}$ as the only answer set, we have that $\emptyset \not\models \Pi^{\emptyset}$, $\{a\} \models \Pi^{\{a\}}$ and $\emptyset \not\models \Pi^{\{a\}}$. Since $\{a\}$ and $f(r)$ are two of such programs Π, and a is the only atom that occurs in r and $f(r)$, we can conclude that $\{r\}$ and $f(r)$ are strongly equivalent by Lemma 1. □

Lemma 5. *For every rule r and fact a, $\{r, a\}$ and $f(r) \cup \{a\}$ have the same answer sets.*

Proof. In view of Lemma 4, $f(r) \cup \{a\}$ and $f(r) \cup f(a)$ have the same answer sets, and the same holds for $\{r, a\}$ and $f(r) \cup f(a)$ by hypothesis. □

Lemma 6. *For every rule r of the form $a \leftarrow a$,*

(i) $\emptyset \models f(r)^{\emptyset}$,
(ii) $\{a\} \models f(r)^{\{a\}}$, *and*
(iii) $\emptyset \models f(r)^{\{a\}}$.

Proof. First of all, since the empty set in the only answer set for $\{r\}$ and then for $f(r)$, (i) is clearly true. Now consider the program consisting of rule r plus fact a. Since $\{a\}$ is an answer set for $\{r, a\}$, it is an answer set for $f(r) \cup \{a\}$ also by Lemma 5. Consequently, $\{a\} \models (f(r) \cup \{a\})^{\{a\}}$, which proves (ii). From (ii) and the fact that $\{a\}$ is not an answer set for $f(r)$, (iii) follows also. □

Lemma 7. *For every rule r of the form $a \leftarrow b$,*

(i) $\emptyset \models f(r)^{\emptyset}$,
(ii) $\{a\} \models f(r)^{\{a\}}$,
(iii) $\emptyset \models f(r)^{\{a\}}$, *and*
(iv) $\{b\} \not\models f(r)^{\{b\}}$.

Proof. The proof of the first three claims is similar to the one of Lemma 6. To prove (iv), consider that since $\{b\}$ is not an answer set for $\{r, b\}$, it is not an answer set for $f(r) \cup \{b\}$ either by Lemma 5. But $\emptyset \not\models (f(r) \cup \{b\})^{\{b\}}$ because $\emptyset \not\models \{b\}^{\{b\}}$; consequently $\{b\} \not\models (f(r) \cup \{b\})^{\{b\}}$. Since $\{b\} \models \{b\}^{\{b\}}$, we can conclude (iv). □

Lemma 8. *For every rule r of the form $a \leftarrow b$,*

(i) $\{a, b\} \models f(r)^{\{a,b\}}$,
(ii) $\{b\} \not\models f(r)^{\{a,b\}}$, *and*
(iii) $\{a\} \models f(r)^{\{a,b\}}$.

Proof. Set $\{a, b\}$ is an answer set for $\{r, b\}$, and consequently for $f(r) \cup \{b\}$ also by Lemma 5. Consequently $\{a, b\} \models (f(r) \cup \{b\})^{\{a,b\}}$ — from which we derive (i) — and all proper subsets of $\{a, b\}$ don't satisfy $(f(r) \cup \{b\})^{\{a,b\}}$. Since $\{b\} \models \{b\}^{\{a,b\}}$, we have that (ii) holds. Notice that $\{a, b\} \models (f(r) \cup \{a\})^{\{a,b\}}$ follows from (i), and that $\{a, b\}$ is not an answer set for $f(r) \cup \{a\}$ because it is not an answer set for $\{r, a\}$ and by Lemma 5. Consequently there is a proper subset of $\{a, b\}$ that satisfies $(f(r) \cup \{a\})^{\{a,b\}}$. Such subset can only be $\{a\}$ because it is the only one that satisfies $\{a\}^{\{a,b\}}$. We can conclude (iii). □

Lemma 9.

$$\emptyset \models f(a \leftarrow b)^{\{a,b\}}.$$

Proof. Let r be $a \leftarrow b$, and r' be $b \leftarrow a$. Lemma 8 can help us determine which subsets of $\{a, b\}$ satisfy $(f(r) \cup f(r'))^{\{a,b\}}$: from part (i) applied to both r and r', we get that $\{a, b\}$ satisfies this program, while $\{b\}$ (by part (ii) applied to r) and $\{a\}$ (by part (ii) applied to r') don't. On the other hand $\{a, b\}$ is not an answer set for $\{r, r'\}$ and then for $f(r) \cup f(r')$ by the soundness hypothesis. We can conclude that $\emptyset \models (f(r) \cup f(r'))^{\{a,b\}}$ from which the lemma's assertion follows. □

Lemma 10. *For every unary program Π, Π and $\bigcup_{r \in \Pi} f(r)$ are strongly equivalent.*

Proof. In view of Lemma 2, it is sufficient to show that for each unary rule r, $\{r\}$ and $f(r)$ are strongly equivalent. For rules that are facts, this is shown by Lemma 4. For rules r of the form $a \leftarrow a$, in view of Lemma 6 it is easy to check that, for every sets X and Y such that $X \subseteq Y \subseteq \{a\}$, $X \models (a \leftarrow a)^Y$ iff $X \models f(a \leftarrow a)^Y$. So $a \leftarrow a$ and $f(a \leftarrow a)$ are strongly equivalent by Lemma 1. Similarly, we can check that for every sets X and Y such that $X \subseteq Y \subseteq \{a, b\}$

with $X \neq \emptyset$ or $Y \neq \{b\}$, that $X \models (a \leftarrow b)^Y$ iff $X \models f(a \leftarrow b)^Y$. (The case with $X = \emptyset$ and $Y = \{b\}$ is irrelevant for strong equivalence since $\{b\} \not\models (a \leftarrow b)^{\{b\}}$.) This is by Lemmas 7–9. Consequently $a \leftarrow b$ and $f(a \leftarrow b)$ are strongly equivalent by Lemma 1 as well. □

Now we are ready to prove the second part of the main theorem: for any rule $r \in \mathrm{Dom}(f)$, $f(r)$ and $\{r\}$ are strongly equivalent. By Lemma 3,it is sufficient to show that, for each unary program Π, $\Pi \cup \{r\}$ and $\Pi \cup f(r)$ have the same answer sets. First we notice that $\Pi \cup \{r\}$ and $\bigcup_{r' \in \Pi \cup \{r\}} f(r')$ have the same answer sets since $\Pi \cup \{r\} \subseteq \mathrm{Dom}(f)$ and for the soundness of the transformation. Then we can see that $\bigcup_{r' \in \Pi \cup \{r\}} f(r') = \bigcup_{r' \in \Pi} f(r') \cup f(r)$. Finally, $\bigcup_{r' \in \Pi} f(r') \cup f(r)$ and $\Pi \cup f(r)$ have the same answer sets because, by Lemma 10, programs Π and $\bigcup_{r' \in \Pi} f(r')$ are strongly equivalent. □

6.2 Proof of Propositions 1 and 2

Proposition 1. *For any program Π, Π and $\bigcup_{r \in \Pi} f_1(r)$ have the same answer sets.*

Proof. (outline) Consider any set of atoms X. If $\{p, q\} \not\subseteq X$ then $f(F \leftarrow G)^X$ is essentially $\{F \leftarrow G\}^X$, and the claim easily follows. Otherwise $f_1(F \leftarrow G)^X$ is essentially $\{F_{p \leftrightarrow q} \leftarrow G_{p \leftrightarrow q}\}$. If we extend the notation of the subscript $p \leftrightarrow q$ to both programs and sets of atoms, $(\bigcup_{r \in \Pi} f_1(r))^X$ can be seen as $(\Pi^X)_{p \leftrightarrow q}$. A subset Y of X satisfies Π^X iff $Y_{p \leftrightarrow q}$ satisfies $(\Pi^X)_{p \leftrightarrow q}$. Since $Y_{p \leftrightarrow q} \subseteq X$ and $|Y_{p \leftrightarrow q}| = |Y|$, we can conclude that X is a minimal set satisfying $(\bigcup_{r \in \Pi} f_1(r))^X$ iff it is a minimal set satisfying Π^X. □

Proposition 2. *For any program $\Pi \subseteq NBR$, Π and $\bigcup_{r \in \Pi} f_2(r)$ have the same answer sets.*

Proof. (outline) Let Π' be $\bigcup_{r \in \Pi} f_2(r)$. The proof is based on the fact that Π and Π' are both absolutely tight. So each set X atoms is an answer set for Π iff X satisfies Π and X is "supported" by Π, and similarly for Π' (for more details, see [Lee, 2005]).

It is not hard to see that the same sets of atoms satisfy Π and Π', and that Π and Π' are supported by the same sets of atoms. □

6.3 Proof of Theorem 2

For simplicity, we consider the definition of an answer set for NDR programs as defined in [Lifschitz and Woo, 1992], in which the reduct Π^X consists of the rule

$$head^+(r) \leftarrow body^+(r) \tag{7}$$

($head^+(r)$ here stands for the disjunction of its elements, $body^+(r)$ for their conjunction) for every rule r in Π such that $head^-(r) \subseteq X$ and $body^-(r) \cap X = \emptyset$.

Since Π^X is satisfied by the same sets of atoms regardless on the definition, the strong equivalent criterion based on satisfaction of the reduct doesn't change.

Let Π be a program strongly equivalent to r. Let X be $head^+(r) \cup head^-(r) \cup body^+(r)$, and let Y be $body^+(r)$. Then r^X is (7). By Lemma 1, since $X \models r^X$ (recall that $head^+(r)$ is nonempty by hypothesis) then $X \models \Pi^X$, and since $Y \not\models r^X$ it follows that $Y \not\models \Pi^X$. Consequently, there is a rule of Π — the rule r' of the theorem's statement — such that $X \models (r')^X$ and $Y \not\models (r')^X$. From this second fact, $(r')^X$ is nonempty so it is

$$head^+(r') \leftarrow body^+(r'), \tag{8}$$

and also $Y \models body^+(r')$ and $Y \not\models head^+(r')$.

To prove that $head^+(r) \subseteq head^+(r')$, take any atom $a \in head^+(r)$. The set $Y \cup \{a\}$ satisfies r^X, so it satisfies Π^X by Lemma 1, and then $(r')^X$ also. On the other hand, since $Y \models body^+(r')$ and $Y \subseteq Y \cup \{a\}$, we have that $Y \cup \{a\} \models body^+(r')$. Consequently, $Y \cup \{a\} \models head^+(r')$. Since $Y \not\models head^+(r')$, we can conclude that a is an element of $head^+(r')$.

The proof that $body^+(r) \subseteq body^+(r')$ is similar to the previous part of the proof, by taking any $a \in body^+(r)$ and considering the set $Y \setminus \{a\}$ instead of $Y \cup \{a\}$.

To prove that $head^-(r) \subseteq head^-(r')$, take any atom $a \in head^-(r)$. Since $r^{X\setminus\{a\}}$ is empty, it is satisfied, in particular, by Y and $X \setminus \{a\}$. Consequently, $Y \models (r')^{X\setminus\{a\}}$ by Lemma 1. On the other hand, $Y \not\models (r')^X$, so $(r')^{X\setminus\{a\}}$ is not (8), and then it is empty. The only case in which $(r')^{X\setminus\{a\}}$ is empty and $(r')^X$ is not is if $a \in head^-(r')$.

The proof that $body^-(r) \subseteq body^-(r')$ is similar to the previous part of the proof, by taking any $a \in body^-(r)$ and considering the reduct $r^{X\cup\{a\}}$.

6.4 Proofs of Propositions 5–7 (Outline)

In the proof of Proposition 5, from SNDR to CCR, we take the following sound translation f: if r has the form (4) then $f(r)$ is

$$1\{a\} \leftarrow 1\{b_1\}, \ldots, 1\{b_n\}, \{c_1\}0, \ldots, \{c_m\}0, \{not\ d_1\}0, \ldots, \{not\ d_q\}0,$$

and, if r has the form (5), then $f(r)$ is

$$1\{\} \leftarrow 1\{b_1\}, \ldots, 1\{b_n\}, \{c_1\}0, \ldots, \{c_m\}0, 1\{d_1\}, \ldots, 1\{d_q\}.$$

A sound translation f from VSNDR to PCCR (proof of Proposition 6) is defined as follows: if r has the form (6) then $f(r)$ is

$$\{a\} \leftarrow 1\{b_1\}, \ldots, 1\{b_n\}, \{c_1\}0, \ldots, \{c_m\}0$$

while for rules r of the form (5), $f(r)$ is the same as in the previous translation.

The proof in the other direction is based on the modular translation from programs with cardinality constraints to programs with nested expressions whose heads are atoms or $\bot$, as defined in [Ferraris and Lifschitz, 2005]. If we first apply such translation to any CCR-program, then the one from RNE to NDR

of [Lifschitz *et al.*, 1999], we get a SNDR-program. Similarly from a PCCR-program we get a VSNDR-program.

For Proposition 7 assume, in sake of contradiction, that a sound translation from CCR to PCCR exists. Then, in view of Propositions 5 and 6, a sound translation from SNDR to VSNDR exists. This is impossible in view of Theorem 2 and the Main Theorem.

7 Conclusions

We have established a relationship between modular transformations and strong equivalence. We showed how it can be used to determine whether sound modular translations between languages are possible.

Other definitions of a modular translation allow the the introduction of auxiliary atoms. This is, for instance, the case for the definitions in [Ferraris, 2005] and [Janhunen, 2000]. These two papers are also different from the work described in this note in that they take into account the computation time of translation algorithms.

We restricted, in Sect. 2, the domain and range of transformations to programs with nested expressions. If we drop this limitation by allowing arbitrary propositional formulas, and we define the soundness of a transformation in terms of equilibrium logic [Pearce, 1997, 1999] then the Main Theorem will still hold. Since each propositional theory is strongly equivalent to a logic program [Cabalar and Ferraris, 2005] we can conclude that there exists a sound and modular translation from propositional theories to RNE and vice versa.

Criteria for strong equivalence, in part related to Theorem 2, are proposed in [Lin and Chen, 2005].

The theorems about cardinality constraints stated in Sect. 4 can be trivially extended to arbitrary weight constraints in view of the fact that an expression $c = w$ in a weight constraint can always be replaced by w copies of $c = 1$.

Acknowledgments

I am grateful to Joohyung Lee for comments on this work. Special thanks go to Vladimir Lifschitz for many comments and discussions on the topic, and his careful reading of this paper. This research was partially supported by the National Science Foundation under Grant IIS-0412907.

References

[Cabalar and Ferraris, 2005] Pedro Cabalar and Paolo Ferraris. Propositional theories are equivalent to logic programs. In preparation, 2005.

[Eiter and Gottlob, 1993] Thomas Eiter and Georg Gottlob. Complexity results for disjunctive logic programming and application to nonmonotonic logics. In Dale Miller, editor, *Proceedings of International Logic Programming Symposium (ILPS)*, pages 266–278, 1993.

[Erdem and Lifschitz, 2003] Esra Erdem and Vladimir Lifschitz. Tight logic programs. *Theory and Practice of Logic Programming*, 3:499–518, 2003.

[Ferraris and Lifschitz, 2005] Paolo Ferraris and Vladimir Lifschitz. Weight constraints as nested expressions. *Theory and Practice of Logic Programming*, 5:45–74, 2005.

[Ferraris, 2005] Paolo Ferraris. A modular, polynomial method for eliminating weight constraints. [2] Unpublished draft, 2005.

[Gelfond and Lifschitz, 1988] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In Robert Kowalski and Kenneth Bowen, editors, *Proceedings of International Logic Programming Conference and Symposium*, pages 1070–1080, 1988.

[Janhunen, 2000] Tomi Janhunen. Comparing the expressive powers of some syntactically restricted classes of logic programs. In *Proc. 1st International Conference on Computational Logic*, volume 1861, pages 852–866, 2000.

[Lee, 2005] Joohyung Lee. A model-theoretic counterpart of loop formulas. In *Proc. IJCAI*, 2005. To appear.

[Lifschitz and Woo, 1992] Vladimir Lifschitz and Thomas Woo. Answer sets in general nonmonotonic reasoning (preliminary report). In Bernhard Nebel, Charles Rich, and William Swartout, editors, *Proc. Third Int'l Conf. on Principles of Knowledge Representation and Reasoning*, pages 603–614, 1992.

[Lifschitz *et al.*, 1999] Vladimir Lifschitz, Lappoon R. Tang, and Hudson Turner. Nested expressions in logic programs. *Annals of Mathematics and Artificial Intelligence*, 25:369–389, 1999.

[Lifschitz *et al.*, 2001] Vladimir Lifschitz, David Pearce, and Agustin Valverde. Strongly equivalent logic programs. *ACM Transactions on Computational Logic*, 2:526–541, 2001.

[Lin and Chen, 2005] Fangzhen Lin and Yin Chen. Discovering classes of strongly equivalent logic programs. In *Proc. IJCAI*, 2005. To appear.

[Marek and Niemelä, 2004] Victor Marek and Ilkka Niemelä. On logic programs with cardinality constraints. In *Proc. 7th Int'l Conference on Logic Programming and Nonmonotonic Reasoning*, pages 154–166, 2004.

[Marek and Truszczyński, 1991] Victor Marek and Mirosław Truszczyński. Autoepistemic logic. *Journal of the ACM*, 38:588–619, 1991.

[Pearce, 1997] David Pearce. A new logical characterization of stable models and answer sets. In Jürgen Dix, Luis Pereira, and Teodor Przymusinski, editors, *Non-Monotonic Extensions of Logic Programming (Lecture Notes in Artificial Intelligence 1216)*, pages 57–70. Springer-Verlag, 1997.

[Pearce, 1999] David Pearce. From here to there: Stable negation in logic programming. In D. Gabbay and H. Wansing, editors, *What Is Negation?* Kluwer, 1999.

[Simons *et al.*, 2002] Patrik Simons, Ilkka Niemelä, and Timo Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138:181–234, 2002.

[Turner, 2003] Hudson Turner. Strong equivalence made easy: nested expressions and weight constraints. *Theory and Practice of Logic Programming*, 3(4,5):609–622, 2003.

[2] `http://www.cs.utexas.edu/users/otto/papers/newweight.ps` .

Guarded Open Answer Set Programming

Stijn Heymans, Davy Van Nieuwenborgh*, and Dirk Vermeir**

Dept. of Computer Science,
Vrije Universiteit Brussel, VUB,
Pleinlaan 2, B1050 Brussels, Belgium
{sheymans,dvnieuwe,dvermeir}@vub.ac.be

Abstract. Open answer set programming (OASP) is an extension of answer set programming where one may ground a program with an arbitrary superset of the program's constants. We define a fixed point logic (FPL) extension of Clark's completion such that open answer sets correspond to models of FPL formulas and identify a syntactic subclass of programs, called (loosely) guarded programs. Whereas reasoning with general programs in OASP is undecidable, the FPL translation of (loosely) guarded programs falls in the decidable (loosely) guarded fixed point logic (μ(L)GF).

Moreover, we reduce normal closed ASP to loosely guarded OASP, enabling a characterization of an answer set semantics by μLGF formulas. Finally, we relate guarded OASP to Datalog LITE, thus linking an answer set semantics to a semantics based on fixed point models of extended stratified Datalog programs. From this correspondence, we deduce 2-EXPTIME-completeness of satisfiability checking w.r.t. (loosely) guarded programs.

1 Introduction

A problem with finite closed answer set programming (ASP)[10] is that all significant constants have to be present in the program in order to capture the intended semantics. E.g., a program with a rule $r : p(X) \leftarrow not\ q(X)$ and a fact $q(a)$ has the unique answer set $\{q(a)\}$ and thus leads to the conclusion that p is not satisfiable. However, if r is envisaged as a schema constraint and a is just one possible data instance, this conclusion is wrong: other data makes p satisfiable.

This problem was solved in [11] by introducing k new constants, k finite, and grounding the program with this extended universe; the answer sets of the grounded program were called *k-belief sets*. We extended this idea, e.g. in [16], by allowing for arbitrary, thus possibly infinite, universes. *Open answer sets* are then pairs (U, M) with M an answer set of the program grounded with U. The above program has an open answer set $(\{x, a\}, \{q(a), p(x)\})$ where p is satisfiable.

Characteristic about (O)ASP is its treatment of negation as failure (naf): one guesses an interpretation for a program, computes the program without naf (the GL-reduct[10]),

* Supported by the FWO.

** This work was partially funded by the Information Society Technologies programme of the European Commission, Future and Emerging Technologies under the IST-2001-37004 WASP project.

C. Baral et al. (Eds.): LPNMR 2005, LNAI 3662, pp. 92–104, 2005.
© Springer-Verlag Berlin Heidelberg 2005

calculates the iterated fixed point of this reduct, and checks whether this fixed point equals the initial interpretation. We compile these external manipulations, i.e. not expressible in the language of programs itself, into fixed point logic (FPL)[14] formulas that are at most quadratic in the size of the original program. First, we rewrite an arbitrary program as a program containing only one designated predicate p and (in)equality; this makes sure that when calculating a fixed point of the predicate variable p, it constitutes a fixed point of the whole program. In the next phase, such a *p-program P* is translated to FPL formulas $\texttt{comp}(P)$. $\texttt{comp}(P)$ ensures satisfiability of program rules by formulas comparable to those in Clark's completion. The specific answer set semantics is encoded by formulas indicating that for each atom $p(\boldsymbol{x})$ in the model there must be a true rule body that motivates the atom, and this in a minimal way, i.e. using a fixed point predicate. Negation as failure is correctly handled by making sure that only those rules that would be present in the GL-reduct can be used to motivate atoms.

In [5], Horn clauses were translated to FPL formulas and in [12] reasoning with an extension of stratified Datalog was reduced to FPL, but, to the best of our knowledge, this is the first encoding of an answer set semantics in FPL.

In [21,19], ASP with (finite) propositional programs is reduced to propositional satisfiability checking. The translation makes the loops in a program explicit and ensures that atoms $p(\boldsymbol{x})$ are motivated by bodies outside of these loops. Although this is an elegant characterization of answer sets in the propositional case, the approach does not seem to hold for OASP, where programs are not propositional but possibly ungrounded and with infinite universes. Instead, we directly use the built-in "loop detection" mechanism of FPL, which enables us to go beyond propositional programs.

Translating OASP to FPL is thus interesting in its own right, but it also enables the analysis of decidability of OASP via decidability results of fragments of FPL. Satisfiability checking of a predicate p w.r.t. a program, i.e. checking whether there exists an open answer set containing some $p(\boldsymbol{x})$, is undecidable, e.g. the undecidable domino problem can be reduced to it[15]. It is well-known that satisfiability checking in FOL is undecidable, and thus the extension to FPL is too. However, expressive decidable fragments of FPL have been identified[14]: *(loosely) guarded fixed point logic* (μ(L)GF) extends the *(loosely) guarded fragment* (L)GF of FOL with fixed point predicates.

GF was identified in [2] as a fragment of FOL satisfying properties such as decidability of reasoning and the tree-model property, i.e. every model can be rewritten as a tree-model. The restriction of quantified variables by a *guard*, an atom containing the variables in the formula, ensures decidability in GF. Guards are responsible for the tree-model property of GF (where the concept of tree is adapted for predicates with arity larger than 2), which in turn enables tree-automata techniques for showing decidability of satisfiability checking. In [4], GF was extended to LGF where guards can be conjunctions of atoms and, roughly, every pair of variables must be together in some atom in the guard. Satisfiability checking in both GF and LGF is 2-EXPTIME-complete[13], as are their extensions with fixed point predicates μGF and μLGF[14].

We identify a syntactically restricted class of programs, *(loosely) guarded programs ((L)GPs)*, for which the FPL translation falls in μ(L)GF, making satisfiability checking w.r.t. (L)GPs decidable and in 2-EXPTIME. In LGPs, rules have a set of atoms, the guard, in the positive body, such that every pair of variables in the rule appears together

in an atom in that guard. GPs are the restriction of LGPs where guards must consist of exactly one atom. Programs under the normal answer set semantics can be rewritten as LGPs under the open answer set semantics by guarding all variables with atoms that can only deduce constants from the original program. Besides the desirable property that OASP with LGPs is thus a proper decidable extension of normal ASP, this yields that satisfiability checking w.r.t. LGPs is, at least, NEXPTIME-hard.

Datalog LITE[12] is a language based on stratified Datalog with input predicates where rules are monadic or guarded and may have generalized literals in the body, i.e. literals of the form $\forall \boldsymbol{Y} \cdot a \Rightarrow b$ for atoms a and b. It has an appropriately adapted bottom-up fixed point semantics. Datalog LITE was devised to ensure linear time model checking while being expressive enough to capture *computational tree logic*[8] and alternation-free μ-calculus[18]. Moreover, it was shown to be equivalent to alternation-free μGF. Our reduction of GPs to μGF, ensures that we have a reduction from GPs to Datalog LITE, and thus couples the answer set semantics to a fixed point semantics based on stratified programs. Intuitively, the guess for an interpretation in the answer set semantics corresponds to the input structure one feeds to the stratified Datalog program. The translation from GPs to Datalog LITE needs only one stratum to subsequently perform the minimality check of answer set programming.

The other way around, we reduce satisfiability checking in recursion-free Datalog LITE to satisfiability checking w.r.t. GPs. Recursion-free Datalog LITE is equivalent to GF[12], and, since satisfiability checking of GF formulas is 2-EXPTIME-hard[13], we obtain 2-EXPTIME-completeness for satisfiability checking w.r.t. (L)GPs.

In [16,17], other decidable classes of programs under the open answer set semantics were identified; decidability was attained differently than for (L)GPs, by reducing OASP to finite ASP. Although the therein identified *conceptual logic programs* are more expressive in some aspects (they allow for a more liberal use of inequality), they are less expressive in others, e.g. the use of predicates is restricted to unary and binary ones. Moreover, the definition of (L)GPs is arguably more simple compared to the often intricate restrictions on the rules in conceptual logic programs.

The remainder of the paper is organized as follows. After recalling the open answer set semantics in Section 2, we reduce reasoning under the open answer set semantics to reasoning with FPL formulas in Section 3. Section 4 describes guarded OASP, together with a 2-EXPTIME complexity upper bound and a reduction from finite ASP to loosely guarded OASP. Section 5 discusses the relationship with Datalog LITE and establishes 2-EXPTIME-completeness for (loosely) guarded open answer set programming. Section 6 contains conclusions and directions for further research. Due to space restrictions, proofs have been omitted; they can be found in [15].

2 Open Answer Set Semantics

We recall the open answer set semantics from [16]. *Constants*, *variables*, *terms*, and *atoms* are defined as usual. A *literal* is an atom $p(\boldsymbol{t})$ or a *naf-atom* $not\ p(\boldsymbol{t})$.[1] The

[1] We have no negation $\neg$, however, programs with $\neg$ can be reduced to programs without it, see e.g. [20].

positive part of a set of literals α is $\alpha^+ = \{p(\boldsymbol{t}) \mid p(\boldsymbol{t}) \in \alpha\}$ and the *negative part* of α is $\alpha^- = \{p(\boldsymbol{t}) \mid not\ p(\boldsymbol{t}) \in \alpha\}$. We assume the existence of binary predicates $=$ and $\neq$, where $t = s$ is considered as an atom and $t \neq s$ as $not\ t = s$. E.g. for $\alpha = \{X \neq Y, Y = Z\}$, we have $\alpha^+ = \{Y = Z\}$ and $\alpha^- = \{X = Y\}$. A *regular* atom is an atom that is not an equality atom. For a set X of atoms, $not\ X = \{not\ l \mid l \in X\}$.

A *program* is a countable set of rules $\alpha \leftarrow \beta$, where α and β are finite sets of literals, $|\alpha^+| \leq 1$, and $\forall t, s \cdot t = s \notin \alpha^+$, i.e. α contains at most one positive atom, and this atom cannot be an equality atom.[2] The set α is the *head* of the rule and represents a disjunction of literals, while β is called the *body* and represents a conjunction of literals. If $\alpha = \emptyset$, the rule is called a *constraint*. *Free rules* are rules of the form $q(\boldsymbol{t}) \vee not\ q(\boldsymbol{t}) \leftarrow$ for a tuple $\boldsymbol{t}$ of terms; they enable a choice for the inclusion of atoms. Atoms, literals, rules, and programs that do not contain variables are *ground*.

For a program P, let $cts(P)$ be the constants in P, $vars(P)$ its variables, and $preds(P)$ its predicates. A *universe* U for P is a non-empty countable superset of the constants in P: $cts(P) \subseteq U$. We call P_U the ground program obtained from P by substituting every variable in P by every possible constant in U. Let $\mathcal{B}_P$ be the set of regular atoms that can be formed from a ground program P.

An *interpretation* I of a ground P is any subset of $\mathcal{B}_P$. For a ground regular atom $p(\boldsymbol{t})$, we write $I \models p(\boldsymbol{t})$ if $p(\boldsymbol{t}) \in I$; For an equality atom $p(\boldsymbol{t}) \equiv t = s$, we have $I \models p(\boldsymbol{t})$ if s and t are equal terms. We have $I \models not\ p(\boldsymbol{t})$ if $I \not\models p(\boldsymbol{t})$. For a set of ground literals X, $I \models X$ if $I \models l$ for every $l \in X$. A ground rule $r : \alpha \leftarrow \beta$ is *satisfied* w.r.t. I, denoted $I \models r$, if $I \models l$ for some $l \in \alpha$ whenever $I \models \beta$, i.e. r is *applied* whenever it is *applicable*. A ground constraint $\leftarrow \beta$ is satisfied w.r.t. I if $I \not\models \beta$. For a ground program P without *not*, an interpretation I of P is a *model* of P if I satisfies every rule in P; it is an *answer set* of P if it is a subset minimal model of P. For ground programs P containing *not*, the *GL-reduct*[10] w.r.t. I is defined as P^I, where P^I contains $\alpha^+ \leftarrow \beta^+$ for $\alpha \leftarrow \beta$ in P, $I \models not\ \beta^-$ and $I \models \alpha^-$. I is an *answer set* of a ground P if I is an answer set of P^I.

In the following, a program is assumed to be a finite set of rules; infinite programs only appear as byproducts of grounding a finite program with an infinite universe. An *open interpretation* of a program P is a pair (U, M) where U is a universe for P and M is an interpretation of P_U. An *open answer set* of P is an open interpretation (U, M) of P with M an answer set of P_U. An n-ary predicate p in P is *satisfiable* if there is an open answer set (U, M) of P and a $\boldsymbol{x} \in U^n$ such that $p(\boldsymbol{x}) \in M$.

3 Open Answer Set Programming via Fixed Point Logic

We assume without loss of generality that the set of constants and the set of predicates in a program are disjoint and that each predicate q has one associated arity, e.g. $q(x)$ and $q(x, y)$ are not allowed. A program P is a *p-program* if p is the only predicate in P different from the (in)equality predicate. We can rewrite any program P as an equivalent p-program P_p by replacing every regular m-ary atom $q(\boldsymbol{t})$ in P by $p(\boldsymbol{t}, \boldsymbol{0}, q)$ where p has arity n, with n the maximum of the arities of predicates in P augmented by

[2] The condition $|\alpha^+| \leq 1$ ensures that the GL-reduct is non-disjunctive.

1, **0** a sequence of new constants 0 of length $n - m - 1$, and q a new constant with the same name as the original predicate. Furthermore, in order to avoid interference of the new constants, we add for every variable X in a rule $r \in P$ and for every newly added constant a in P_p, $X \neq a$ to the body. E.g., the rule $h(a, b) \leftarrow q(X)$ in P corresponds to $p(a, b, h) \leftarrow p(X, 0, q), X \neq 0, X \neq h, X \neq q$ in P_p.

Proposition 1. *An open interpretation (U, M) is an open answer set of P iff $(U \cup preds(P) \cup \{0\}, \{p(\boldsymbol{x}, \mathbf{0}, q) \mid q(\boldsymbol{x}) \in M\})$ is an open answer set of the p-program P_p.*

The translation of a program to a p-program does not influence the complexity of reasoning, i.e. the size of P_p is linear in the size of P. By Proposition 1, we can focus attention on p-programs only. Since p-programs have open answer sets consisting of one predicate p, fixed points calculated w.r.t. p yield minimal models of the whole program as we will show in Proposition 2.

In [5], a similar motivation drives the reduction of Horn clauses to clauses consisting of only one defined predicate. Their encoding does not introduce new constants to identify old predicates and depends entirely on the use of (in)equality. However, to account for databases consisting of only one element, [5] needs an additional transformation that unfolds bodies of clauses.

We assume that FOL interpretations have the same form as open interpretations: a pair (U, M) corresponds with the FOL interpretation M over the domain U. Furthermore, we consider FOL with equality such that equality is always interpreted as the identity relation over U. *(Least) Fixed Point Logic (FPL)* is defined along the lines of [14]. *Fixed point formulas* are of the form

$$[\text{LFP } W\boldsymbol{X}.\psi(W, \boldsymbol{X})](\boldsymbol{X}) \ , \tag{1}$$

where W is an n-ary predicate variable, $\boldsymbol{X}$ is an n-ary sequence of variables, $\psi(W, \boldsymbol{X})$ is a FOL formula where all free variables are contained in $\boldsymbol{X}$ and where W appears only positively in $\psi(W, \boldsymbol{X})$.[3]

We associate with (1) and an interpretation (U, M) that does not interpret W, an operator $\psi^{(U,M)} : 2^{U^n} \rightarrow 2^{U^n}$ defined on sets S of n-ary tuples as $\psi^{(U,M)}(S) \equiv \{\boldsymbol{x} \in U^n \mid (U, M) \models \psi(S, \boldsymbol{x})\}$. By definition, W appears only positively in ψ such that $\psi^{(U,M)}$ is monotonic on sets of n-ary U-tuples and has a least fixed point, which we denote by LFP$(\psi^{(U,M)})$. Finally, we have $(U, M) \models [\text{LFP } W\boldsymbol{X}.\psi(W, \boldsymbol{X})](\boldsymbol{x})$ iff $\boldsymbol{x} \in \text{LFP}(\psi^{(U,M)})$.

We can reduce a p-program P to equivalent FPL formulas $\texttt{comp}(P)$. The *completion* $\texttt{comp}(P)$ consists of formulas $a \neq b$ for different constants a and b in P making sure that constants are interpreted as different elements, where $a \neq b \equiv \neg(a = b)$. $\texttt{comp}(P)$ also contains the formula $\exists X \cdot \textbf{true}$ ensuring the existence of at least one element in the domain of an interpretation. Besides these technical requirements that match FOL interpretations with open interpretations, $\texttt{comp}(P)$ contains the formulas in $\texttt{fix}(P) \equiv \texttt{sat}(P) \cup \texttt{gl}(P) \cup \texttt{fpf}(P)$, which can be intuitively categorized as follows: $\texttt{sat}(P)$

[3] Since $\psi(W, \boldsymbol{X})$ is a FOL formula, we do not allow nesting of fixed point formulas. This restriction is sufficient for the FPL simulation of OASP, and, furthermore, it simplifies the notation since one does not have to take into account an extra function χ that gives meaning to free second-order variables different from W.

ensures that a model of $\mathtt{fix}(P)$ satisfies all rules in P, $\mathtt{gl}(P)$ is an auxiliary component defining atoms that indicate when a rule in P belongs to the GL-reduct of P, and finally, $\mathtt{fpf}(P)$ ensures that every model of $\mathtt{fix}(P)$ is a minimal model of the GL-reduct in P; it uses the atoms defined in $\mathtt{gl}(P)$ to select, for the calculation of the fixed point, only those rules in P that are in the GL-reduct of P.

We interpret a naf-atom *not* a in a FOL formula as the literal $\neg a$. Moreover, we assume that, if a set X is empty, $\bigwedge X = \textbf{true}$ and $\bigvee X = \textbf{false}$. We further assume that the arity of p, the only predicate in a p-program, is n.

Definition 1. *Let P be a p-program. Then, $\mathtt{fix}(P) \equiv \mathtt{sat}(P) \cup \mathtt{gl}(P) \cup \mathtt{fpf}(P)$, where*

– $\mathtt{sat}(P)$ *contains formulas*

$$\forall \boldsymbol{Y} \cdot \bigwedge \beta \Rightarrow \bigvee \alpha \tag{2}$$

for rules $\alpha \leftarrow \beta \in P$ with variables $\boldsymbol{Y}$,

– $\mathtt{gl}(P)$ *contains formulas*

$$\forall \boldsymbol{Y} \cdot r(\boldsymbol{Y}) \Leftrightarrow \bigwedge \alpha^- \wedge \bigwedge \neg \beta^- \tag{3}$$

for rules $r : \alpha \leftarrow \beta \in P$ with variables $\boldsymbol{Y}$ and a new predicate r,

– $\mathtt{fpf}(P)$ *contains the formula*

$$\forall \boldsymbol{X} \cdot p(\boldsymbol{X}) \Rightarrow [\mathrm{LFP}\ W\boldsymbol{X}.\phi(W, \boldsymbol{X})](\boldsymbol{X}) \tag{4}$$

with $\phi(W, \boldsymbol{X}) \equiv W(\boldsymbol{X}) \vee \bigvee_{r:p(\boldsymbol{t})\vee\alpha\leftarrow\beta\in P} E(r)$ and $E(r) \equiv \exists \boldsymbol{Y} \cdot X_1 = t_1 \wedge \ldots \wedge X_n = t_n \wedge \bigwedge \beta^+[p|W] \wedge r(\boldsymbol{Y})$, where $\boldsymbol{X} = X_1, \ldots, X_n$ are n new variables, $\boldsymbol{Y}$ are the variables in r, W is a new (second-order) variable and $\beta^+[p|W]$ is β^+ with p replaced by W.

The completion *of P is* $\mathtt{comp}(P) \equiv \mathtt{fix}(P) \cup \{a \neq b \mid a \neq b \in cts(P)\} \cup \{\exists X \cdot \boldsymbol{true}\}$.

The predicate W appears only positively in $\phi(W, \boldsymbol{X})$ such that the fixed point formula in (4) is well-defined. The first conjunct, $W(\boldsymbol{X})$, in $\phi(W, \boldsymbol{X})$ ensures that previously deduced tuples are deduced by the next application of the fixed point operator, i.e. $S \subseteq \phi^{(U,M)}(S)$. The disjunction $\bigvee_r E(r)$ makes sure that for each atom there is a rule r in the GL-reduct ($\exists \boldsymbol{Y} \cdot r(\boldsymbol{Y})$) with a true positive body that can motivate that atom.

Example 1. Take a p-program P with rule $r : p(X) \leftarrow p(X)$. $\mathtt{comp}(P)$ is then such that $\mathtt{sat}(P) = \{\forall X \cdot p(X) \Rightarrow p(X)\}$, ensuring that r is satisfied, and $\mathtt{gl}(P) = \{\forall X \cdot r(X) \Leftrightarrow \textbf{true}\}$ says that r belongs to every GL-reduct since there are no naf-atoms. Finally, $\mathtt{fpf}(P) = \{\forall X_1 \cdot p(X_1) \Rightarrow [\mathrm{LFP}\ WX_1.\phi(W, X_1)](X_1)\}$, with $\phi(W, X_1) \equiv W(X_1) \vee \exists X \cdot X_1 = X \wedge W(X) \wedge r(X)$.

Proposition 2. *Let P be a p-program. Then, (U, M) is an open answer set of P iff $(U, M \cup R)$ is a model of* $\mathtt{comp}(P)$*, where $R \equiv \{r(\boldsymbol{y}) \mid r[\boldsymbol{Y}|\boldsymbol{y}] \in P_U^M, vars(r) = \boldsymbol{Y}\}$, i.e. the atoms corresponding to rules in the GL-reduct of P_U w.r.t. M.*[4]

[4] We denote the substitution of $\boldsymbol{Y} = Y_1, \ldots, Y_d$ with $\boldsymbol{y} = y_1, \ldots, y_d$ in a rule r by $r[\boldsymbol{Y}|\boldsymbol{y}]$.

Example 2. For a universe $U = \{x\}$, we have the unique open answer set $(U, \emptyset)$ of P in Example 1. Since U is non-empty, every open answer set with a universe U satisfies $\exists X \cdot \mathbf{true}$. Both $(U, M_1 = \{p(x), r(x)\})$ and $(U, M_2 = \{r(x)\})$ satisfy $\mathtt{sat}(P) \cup \mathtt{gl}(P)$. Since $\mathrm{LFP}(\phi^{(U,M_1)}) = \mathrm{LFP}(\phi^{(U,M_2)}) = \emptyset$, only (U, M_2) satisfies $\mathtt{fpf}(P)$; (U, M_2) corresponds exactly to the open answer set $(U, \emptyset)$ of P.

The completion in Definition 1 differs from Clark's completion[6] both in the presence of the fixed point construct in (4) and the atoms representing membership of the GL-reduct. For p-programs P, Clark's Completion $\mathtt{ccomp}(P)$ does not contain $\mathtt{gl}(P)$, and $\mathtt{fpf}(P)$ is replaced by the formula $\forall \boldsymbol{X} \cdot p(\boldsymbol{X}) \Rightarrow \bigvee_{r:p(\boldsymbol{t}) \vee \alpha \leftarrow \beta \in P} D(r)$ with $D(r) \equiv \exists \boldsymbol{Y} \cdot X_1 = t_1 \wedge \ldots \wedge X_n = t_n \wedge \bigwedge \beta \wedge \bigwedge \alpha^-$. Program P in Example 1 is the OASP version of the classical example $p \leftarrow p$[19], for which there are FOL models of $\mathtt{ccomp}(P)$ that do not correspond to any answer sets: both $(\{x\}, \{p(x)\})$ and $(\{x\}, \emptyset)$ are FOL models while only the latter is an open answer set of P.

Using Propositions 1 and 2, we can reduce satisfiability checking in OASP to satisfiability checking in FPL. Moreover, with c the number of constants in a program P, the number of formulas $a \neq b$ is $\frac{1}{2}c(c-1)$, and, since the rest of $\mathtt{comp}(P)$ is linear in P, this yields a quadratic bound for the size of $\mathtt{comp}(P)$.

Theorem 1. *Let P be a program and q an n-ary predicate in P. q is satisfiable w.r.t. P iff $p(\boldsymbol{X}, \mathbf{0}, q) \wedge \mathtt{comp}(P_p)$ is satisfiable. Moreover, this reduction is quadratic.*

4 Guarded Open Answer Set Programming

We repeat the definitions of the *loosely guarded fragment*[4] of FOL as in [14]: *The loosely guarded fragment LGF of FOL is defined inductively as follows:*

(1) *Every relational atomic formula belongs to LGF.*
(2) *LGF is closed under propositional connectives $\neg$, $\wedge$, $\vee$, $\Rightarrow$, and $\Leftrightarrow$.*
(3) *If $\psi(\boldsymbol{X}, \boldsymbol{Y})$ is in LGF, and $\alpha(\boldsymbol{X}, \boldsymbol{Y}) = \alpha_1 \wedge \ldots \wedge \alpha_m$ is a conjunction of atoms, then the formulas*

$$\exists \boldsymbol{Y} \cdot \alpha(\boldsymbol{X}, \boldsymbol{Y}) \wedge \psi(\boldsymbol{X}, \boldsymbol{Y})$$
$$\forall \boldsymbol{Y} \cdot \alpha(\boldsymbol{X}, \boldsymbol{Y}) \Rightarrow \psi(\boldsymbol{X}, \boldsymbol{Y})$$

belong to LGF (and $\alpha(\boldsymbol{X}, \boldsymbol{Y})$ is the guard of the formula), provided that free$(\psi) \subseteq$ free$(\alpha) = \boldsymbol{X} \cup \boldsymbol{Y}$ and for every quantified variable $Y \in \boldsymbol{Y}$ and every variable $Z \in \boldsymbol{X} \cup \boldsymbol{Y}$ there is at least one atom α_j that contains both Y and Z (where free(ψ) are the free variables of ψ).

The *loosely guarded (least) fixed point logic* μLGF is LGF extended with fixed point formulas (1) where $\psi(W, \boldsymbol{X})$ is a LGF formula[5] such that W does not appear in guards. The *guarded fragment* GF is defined as LGF where the guards are atoms instead of conjunctions of atoms. The *guarded fixed point logic* μGF is GF extended with fixed point formulas where $\psi(W, \boldsymbol{X})$ is a GF formula such that W does not appear in guards.

[5] Thus, in accordance with our definition of FPL, nesting of (guarded) fixed point logic formulas is not allowed.

Definition 2. *A rule* $r : \alpha \leftarrow \beta$ *is* loosely guarded *if there is a* $\gamma_b \subseteq \beta^+$ *such that every two variables* X *and* Y *from* r *appear together in an atom from* γ_b*; we call* γ_b *a* body guard *of* r*. It is* fully loosely guarded *if it is loosely guarded and there is a* $\gamma_h \subseteq \alpha^-$ *such that every two variables* X *and* Y *from* r *appear together in an atom from* γ_h*;* γ_h *is called a* head guard *of* r*.*

A program P *is a* (fully) loosely guarded program ((F)LGP) *if every non-free rule in* P *is (fully) loosely guarded.*

Example 3. The rule in Example 1 is loosely guarded but not fully loosely guarded. A rule $a(Y) \vee not\ g(X,Y) \leftarrow not\ b(X), f(X,Y)$ has body guard $\{f(X,Y)\}$ and head guard $\{g(X,Y)\}$, and is thus fully loosely guarded.

Definition 3. *A rule is* guarded *if it is loosely guarded with a singleton body guard. It is* fully guarded *if it is fully loosely guarded with singleton body and head guards.*

A program P *is a* (fully) guarded program ((F)GP) *if every non-free rule in* P *is (fully) guarded.*

Every F(L)GP is a (L)GP, and we can rewrite every (L)GP as a F(L)GP.

Example 4. The rule $p(X) \leftarrow p(X)$ can be rewritten as $p(X) \vee not\ p(X) \leftarrow p(X)$ where the body guard is added to the negative part of the head to function as the head guard. Both programs are equivalent: for a universe U, both have the unique open answer set $(U, \emptyset)$.

Formally, we can rewrite every (L)GP P as an equivalent F(L)GP P^{f}, where P^{f} is P with every $\alpha \leftarrow \beta$ replaced by $\alpha \cup not\ \beta^+ \leftarrow \beta$. The body guard of a rule in a (loosely) guarded program P is then also a head guard of the corresponding rule in P^{f}, and P^{f} is indeed a fully (loosely) guarded program.

A rule is vacuously satisfied if the body of a rule in P^{f} is false and consequently the head does not matter; if the body is true then the newly added part in the head becomes false and the rule in P^{f} reduces to its corresponding rule in P.

Proposition 3. *Let* P *be a program. An open interpretation* (U, M) *of* P *is an open answer set of* P *iff* (U, M) *is an open answer set of* P^{f}*.*

Since we copy the positive bodies to the heads, the size of P^{f} only increases linearly in the size of P. Furthermore, the construction of a p-program retains the guardedness properties: P is a (F)LGP iff P_p is a (F)LGP. A similar property holds for (F)GPs.

For a fully (loosely) guarded p-program P, we can rewrite $\mathtt{comp}(P)$ as the equivalent μ(L)GF formulas $\mathtt{gcomp}(P)$. $\mathtt{gcomp}(P)$ is $\mathtt{comp}(P)$ with the following modifications:

- Formula $\exists X \cdot \mathbf{true}$ is replaced by $\exists X \cdot X = X$, a formula guarded by $X = X$.
- Formula (2) is removed if $r : \alpha \leftarrow \beta$ is free and otherwise replaced by

$$\forall \boldsymbol{Y} \cdot \bigwedge \gamma_b \Rightarrow \bigvee \alpha \vee \bigvee \neg(\beta^+ \backslash \gamma_b) \vee \bigvee \beta^- ,$$

  where γ_b is a body guard of r; we logically rewrite formula (2) such that it is (loosely) guarded. If r is a free rule of the form $q(\boldsymbol{t}) \vee not\ q(\boldsymbol{t}) \leftarrow$, we have $\forall Y \cdot \mathbf{true} \Rightarrow q(\boldsymbol{t}) \vee \neg q(\boldsymbol{t}) \in \mathtt{comp}(P)$, which is always true and can be removed.

- Formula (3) is replaced by the formulas $\forall \boldsymbol{Y} \cdot r(\boldsymbol{Y}) \Rightarrow \bigwedge \alpha^- \wedge \bigwedge \neg\beta^-$ and $\forall \boldsymbol{Y} \cdot \bigwedge \gamma_h \Rightarrow r(\boldsymbol{Y}) \vee \bigvee \beta^- \vee \bigvee \neg(\alpha^- \backslash \gamma_h)$, where γ_h is a head guard of r. We thus rewrite an equivalence as two implications. The first implication is guarded by $r(\boldsymbol{Y})$ and the second one is (loosely) guarded by the head guard of the rule – hence the need for a fully (loosely) guarded program, instead of just a (loosely) guarded one.
- For every $E(r)$ in (4), define $T \equiv \{t_i \notin cts(P) \mid 1 \leq i \leq n\}$, and replace $E(r)$ by
$$E'(r) \equiv \bigwedge_{t_i \notin T} X_i = t_i \wedge \exists \boldsymbol{Z} \cdot (\bigwedge \beta^+[p|W] \wedge r(\boldsymbol{Y}))[t_i \in T|X_i] ,$$
with $\boldsymbol{Z} = \boldsymbol{Y} \backslash T$, i.e. move all $X_i = t_i$ where t_i is constant out of the quantifier's scope, and remove the others by substituting each t_i in $\bigwedge \beta^+[p|W] \wedge r(\boldsymbol{Y})$ by X_i. This rewriting makes sure that every variable in the quantified part of $E'(R)$ is guarded by $r(\boldsymbol{Y})[t_i \in T|X_i]$.

Example 5. For the fully guarded p-program P containing a rule $p(X) \vee not\ p(X) \leftarrow p(X)$ with body and head guard $\{p(X)\}$, one has that $\mathtt{sat}(P) = \{\forall X \cdot p(X) \Rightarrow p(X) \vee \neg p(X)\}$, $\mathtt{gl}(P) = \{\forall X \cdot r(X) \Leftrightarrow p(X)\}$ and the formula $\phi(W, X_1)$ in $\mathtt{fpf}(P)$ is $\phi(W, X_1) \equiv W(X_1) \vee \exists X \cdot X_1 = X \wedge W(X) \wedge r(X)$. $\mathtt{gcomp}(P)$ does not modify $\mathtt{sat}(P)$ and rewrites the equivalence in $\mathtt{gl}(P)$ as two guarded implications. The rewritten $\phi(W, X_1)$ is $W(X_1) \vee (W(X_1) \wedge r(X_1))$.

For a fully (loosely) guarded p-program P, $\mathtt{gcomp}(P)$ is a μ(L)GF formula, and it is logically equivalent to $\mathtt{comp}(P)$, i.e. (U, M) is a model of $\mathtt{comp}(P)$ iff (U, M) is a model of $\mathtt{gcomp}(P)$. $\mathtt{gcomp}(P)$ is a simple logical rewriting of $\mathtt{comp}(P)$, with a size linear in the size of $\mathtt{comp}(P)$. Using Proposition 3 and Theorem 1, satisfiability checking w.r.t. (L)GPs can be quadratically reduced to satisfiability checking of a μ(L)GF formula.

Theorem 2. *Let P be a (L)GP and q an n-ary predicate in P. q is satisfiable w.r.t. P iff $p(\boldsymbol{X}, \boldsymbol{0}, q) \wedge \mathtt{gcomp}((P^{\mathrm{f}})_p)$ is satisfiable. Moreover, this reduction is quadratic.*

Since satisfiability checking for μ(L)GF is 2-EXPTIME-complete (Proposition [1.1] in [14]), we have the following upper complexity bound.

Theorem 3. *Satisfiability checking w.r.t. (L)GPs is in 2-*EXPTIME.

An answer set of a program P (in contrast with an *open* answer set) is defined as an answer set of the grounding of P with its own constants, i.e. M is an answer set of P if it is a minimal model of $P^M_{cts(P)}$. As is common in literature, we assume P contains at least one constant.

We can make any program loosely guarded and reduce the answer set semantics for programs to the open answer set semantics for loosely guarded programs. For a program P, let P^{g} be the program P, where for each rule r in P and for each pair of variables X and Y in r, $g(X, Y)$ is added to the body of r. Furthermore, P^{g} contains rules $g(a, b) \leftarrow$ for every $a, b \in cts(P)$, making its size quadratic in the size of P. Note that we assume w.l.o.g. that P does not contain a predicate g.

The newly added guards in the bodies of rules together with the definition of those guards for constants only ensure a correspondence between answer sets and open answer sets where the universe of the latter equals the constants in the program.

Proposition 4. *Let P be a program. M is an answer set of P iff $(cts(P), M \cup \{g(a,b) \mid a, b \in cts(P)\})$ is an open answer set of P^{g}. Moreover, this reduction is quadratic.*

By construction, P^{g} is loosely guarded. We can reduce checking whether there exists an answer set containing a literal to satisfiability checking w.r.t. the open answer set semantics for loosely guarded programs.

Proposition 5. *Let P be a program and q an n-ary predicate in P. There is an answer set M of P with $q(\boldsymbol{a}) \in M$ iff q is satisfiable w.r.t. P^{g}. Moreover, this reduction is quadratic.*

The "only if" direction is trivial; the other direction uses that for every open answer set (U, M') of a loosely guarded program P^{g}, M' contains only terms from $cts(P)$, and can be rewritten as an open answer set $(cts(P), M \cup \{g(a,b) \mid a, b \in cts(P)\})$, after which Proposition 4 becomes applicable.

By [7,3] and the disjunction-freeness of the GL-reduct of the programs we consider, we have that checking whether there exists an answer set M of P containing a $q(\boldsymbol{a})$ is NEXPTIME-complete. Thus, by Proposition 5, satisfiability checking w.r.t. a LGP is NEXPTIME-hard. In the next section, we improve on this result and show that both satisfiability checking w.r.t. GPs and w.r.t. LGPs is actually 2-EXPTIME-hard.

5 Relationship with Datalog LITE

We define *Datalog* LITE as in [12]. A *Datalog rule* is a rule $\alpha \leftarrow \beta$ where $\alpha = \{a\}$ for some atom a. A *basic Datalog program* is a finite set of Datalog rules such that no head predicate appears in negative bodies of rules. Predicates that appear only in the body of rules are *extensional* or *input* predicates. Note that equality is, by the definition of rules, never a head predicate and thus always extensional. The semantics of a basic Datalog program P, given a relational input structure $\mathcal{U}$ defined over extensional predicates of P[6], is given by its *fixed point model*, see e.g. [1]; for a query (P, q), where P is a basic Datalog program and q is a n-ary predicate, we write $\boldsymbol{a} \in (P, q)(\mathcal{U})$ if there is a fixed point model M of P with input $\mathcal{U}$ such that $q(\boldsymbol{a}) \in M$. We call (P, q) satisfiable if there exists a $\mathcal{U}$ and an $\boldsymbol{a}$ such that $\boldsymbol{a} \in (P, q)(\mathcal{U})$.

A program P is a *stratified Datalog program* if it can be written as a union of basic Datalog programs $(P_0, \ldots, P_n)$, so-called *strata*, such that each of the head predicates in P is a head predicate in exactly one stratum P_i. Furthermore, if a head predicate in P_i is an extensional predicate in P_j, then $i < j$. This definition entails that head predicates in the positive body of rules are head predicates in the same or a lower stratum, and head predicates in the negative body are head predicates in a lower stratum. The semantics

[6] We assume that, if $\mathcal{U}$ defines equality, it does so as the identity relation on, at least, the terms in the regular atoms of $\mathcal{U}$ and on the constants in P. Moreover, $\mathcal{U}$ may define equality even if no (in)equality is present in P; one can thus introduce arbitrary universes.

of stratified Datalog programs is defined stratum per stratum, starting from the lowest stratum and defining the extensional predicates on the way up.

A *generalized literal* is of the form $\forall Y_1, \ldots, Y_n \cdot a \Rightarrow b$ where a and b are atoms and $vars(b) \subseteq vars(a)$. A *Datalog* LITE program is a stratified Datalog program, possibly containing generalized literals in the positive body, where each rule is *monadic* or *guarded*. A rule is monadic if each of its (generalized) literals contains only one (free) variable; it is guarded if there exists an atom in the positive body that contains all variables (free variables in the case of generalized literals) of the rule. The definition of stratified is adapted for generalized literals: for a $\forall Y_1, \ldots, Y_n \cdot a \Rightarrow b$ in the body of a rule where the underlying predicate of a is a head predicate, this head predicate must be a head predicate in a lower stratum (i.e. a is treated as a naf-atom) and a head predicate underlying b must be in the same or a lower stratum (i.e. b is treated as an atom). The semantics can be adapted accordingly since a is completely defined in a lower stratum.

In [12], Theorem 8.5., a Datalog LITE query (π_φ, q_φ) was defined for an alternation-free[7] μGF sentence[8] φ such that $(U, M) \models \varphi$ iff $(\pi_\varphi, q_\varphi)(M \cup id(U))$ evaluates to true, where the latter means that q_φ is in the fixed point model of π_φ with input $M \cup id(U)$, and where $id(U) \equiv \{x = x \mid x \in U\}$. For the formal details of this reduction, we refer to [12].

Satisfiability checking with GPs can be polynomially reduced to satisfiability checking in Datalog LITE. Indeed, by Theorem 2, q is satisfiable w.r.t. a GP P iff $p(\boldsymbol{X}, \boldsymbol{0}, q) \wedge \texttt{gcomp}((P^{\mathrm{f}})_p)$ is satisfiable, and the latter is satisfiable iff $\varphi \equiv \exists \boldsymbol{X} \cdot p(\boldsymbol{X}, \boldsymbol{0}, q) \wedge \texttt{gcomp}((P^{\mathrm{f}})_p)$ is. Since φ is a μGF sentence, we have that φ is satisfiable iff (π_φ, q_φ) is satisfiable. By Theorem 2, the translation of P to φ is quadratic in the size of P and the query (π_φ, q_φ) is quadratic in φ[12], resulting in a polynomial reduction.

Theorem 4. *Let P be a GP, q an n-ary predicate in P and φ the μGF sentence $\exists \boldsymbol{X} \cdot p(\boldsymbol{X}, \boldsymbol{0}, q) \wedge \texttt{gcomp}((P^{\mathrm{f}})_p)$. q is satisfiable w.r.t. P iff (π_φ, q_φ) is satisfiable. Moreover, this reduction is polynomial.*

Satisfiability checking in stratified Datalog under the fixed point model semantics can be linearly reduced to satisfiability checking w.r.t. programs under the open answer set semantics. For a stratified Datalog program P, let P^{o} be the program P with free rules $f(\boldsymbol{X}) \vee not\ f(\boldsymbol{X}) \leftarrow$ added for all predicates f that are extensional in the entire program P (with the exception of equality predicates). The free rules in P^{o} mimic the role of extensional predicates from the original P: they allow for an initial free choice of the relational input structure.

Proposition 6. *Let P be a stratified Datalog query (P, q). (P, q) is satisfiable iff q is satisfiable w.r.t. P^{o}. Moreover, this reduction is linear.*

Recursion-free stratified Datalog is stratified Datalog where the head predicates in the positive bodies of rules must be head predicates in a lower stratum. We call recursion-free Datalog LITE where all rules are guarded, i.e. without monadic rules that are not guarded, Datalog LITER, where the definition of recursion-free is appropriately extended to take into account the generalized literals.

---

[7] Since we did not allow nested least fixed point formulas in our definition of μ(L)GF, it is trivially alternation-free.

[8] A sentence is a formula without free variables.

For a Datalog LITER program P, let $\neg\neg P$ be the program P where all generalized literals are replaced by a double negation. E.g. $q(X) \leftarrow f(X), \forall Y \cdot r(X, Y) \Rightarrow s(Y)$ is rewritten as the rules $q(X) \leftarrow f(X), not\ q'(X)$ and $q'(X) \leftarrow r(X, Y), not\ s(Y)$. As indicated in [12], $\neg\neg P$ is equivalent to P and the recursion-freeness ensures that $\neg\neg P$ is stratified. Clearly, $(\neg\neg P)^{\mathrm{o}}$ is a GP.

For a Datalog LITER query (P, q), $(\neg\neg P, q)$ is an equivalent stratified Datalog query. Hence, by Proposition 6, $(\neg\neg P, q)$ is satisfiable iff q is satisfiable w.r.t. $(\neg\neg P)^{\mathrm{o}}$. This reduction is linear since $\neg\neg P$ is linear in the size of P and so is $(\neg\neg P)^{\mathrm{o}}$. Thus satisfiability checking of Datalog LITER queries can be linearly reduced to satisfiability checking w.r.t. GPs.

Theorem 5. *Let (P, q) be a Datalog* LITER *query. (P, q) is satisfiable iff q is satisfiable w.r.t. $(\neg\neg P)^{\mathrm{o}}$. Moreover, this reduction is linear.*

The reduction from μGF sentences φ to Datalog LITE queries (π_φ, q_φ) specializes, as noted in [12], to a reduction from GF sentences to recursion-free Datalog LITE queries. Moreover, the reduction contains only guarded rules such that GF sentences φ are actually translated to Datalog LITER queries (π_φ, q_φ).

Satisfiability checking in the guarded fragment GF is 2-EXPTIME-complete[13], such that, using Theorem 5 and the intermediate Datalog LITER translation, we have that satisfiability checking w.r.t. GPs is 2-EXPTIME-hard. Completeness readily follows from the 2-EXPTIME membership in Theorem 3.

Every GP is a LGP and satisfiability checking w.r.t. to the former is 2-EXPTIME-complete, thus satisfiability checking w.r.t. LGPs is 2-EXPTIME-hard. Completeness follows again from Theorem 3.

Theorem 6. *Satisfiability checking w.r.t. (L)GPs is 2-*EXPTIME*-complete.*

6 Conclusions and Directions for Further Research

We embedded OASP in FPL and used this embedding to identify (loosely) guarded OASP, a decidable fragment of OASP. Finite ASP was reduced to loosely guarded OASP and the relationship with Datalog LITE was made explicit. Finally, satisfiability checking w.r.t. (loosely) guarded OASP was shown to be 2-EXPTIME-complete.

We plan to further exploit the correspondence between (loosely) guarded OASP and μ(L)GF by seeking to apply implementation techniques used for μ(L)GF satisfiability checking directly to (loosely) guarded OASP. Possibly, we can take advantage of the fact that the embedding does not seem to need the full power of μ(L)GF – there are, e.g. , no nested fixed point formulas in the FPL translation of OASP. It is interesting to search for fragments of guarded OASP that can be implemented using existing answer set solvers such as DLV[9] or SMODELS[23]. Another promising direction is to study generalized literals in the context of the answer set semantics: what is an appropriate semantics in the absence of stratification, can this still be embedded in FPL?

Finally, ω-restricted programs[22] are programs where function symbols are allowed but reasoning is kept decidable by "guarding" variables in a rule with a predicate that is in a lower stratification than the predicate of the head of that rule. Since reasoning

with ω-restricted programs is 2-NEXPTIME-complete, it should be possible to simulate guarded open answer set programming in this framework.

References

1. S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.
2. H. Andréka, I. Németi, and J. Van Benthem. Modal Languages and Bounded Fragments of Predicate Logic. *J. of Philosophical Logic*, 27(3):217–274, 1998.
3. C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge Press, 2003.
4. J. Van Benthem. Dynamic Bits and Pieces. In *ILLC research report*. University of Amsterdam, 1997.
5. A. K. Chandra and D. Harel. Horn Clauses and the Fixpoint Query Hierarchy. In *Proc. of PODS '82*, pages 158–163. ACM Press, 1982.
6. K. L. Clark. Negation as Failure. In *Readings in Nonmonotonic Reasoning*, pages 311–325. Kaufmann, 1987.
7. E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov. Complexity and Expressive Power of Logic Programming. *ACM Comput. Surv.*, 33(3):374–425, 2001.
8. E. A. Emerson and E. M. Clarke. Using Branching Time Temporal Logic to Synthesize Synchronization Skeletons. *Sciene of Computer Programming*, 2(3):241–266, 1982.
9. W. Faber, N. Leone, and G. Pfeifer. Pushing Goal Derivation in DLP Computations. In *Proc. of LPNMR*, volume 1730 of *LNCS*, pages 177–191. Springer, 1999.
10. M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programming. In *Proc. of ICLP'88*, pages 1070–1080, Cambridge, Massachusetts, 1988. MIT Press.
11. M. Gelfond and H. Przymusinska. Reasoning in Open Domains. In *Logic Programming and Non-Monotonic Reasoning*, pages 397–413. MIT Press, 1993.
12. G. Gottlob, E. Grädel, and H. Veith. Datalog LITE: A deductive query language with linear time model checking. *ACM Transactions on Computational Logic*, 3(1):1–35, 2002.
13. E. Grädel. On the Restraining Power of Guards. *Journal of Symbolic Logic*, 64(4):1719–1742, 1999.
14. E. Grädel and I. Walukiewicz. Guarded Fixed Point Logic. In *Proc. of LICS '99*, pages 45–54. IEEE Computer Society, 1999.
15. S. Heymans, D. Van Nieuwenborgh, and D. Vermeir. Guarded Open Answer Set Programming. Technical report. http://tinf2.vub.ac.be/˜sheymans/ tech/guarded-oasp.ps.gz.
16. S. Heymans, D. Van Nieuwenborgh, and D. Vermeir. Semantic Web Reasoning with Conceptual Logic Programs. In *Proc. of RuleML 2004*, pages 113–127. Springer, 2004.
17. S. Heymans, D. Van Nieuwenborgh, and D. Vermeir. Nonmonotonic Ontological and Rule-Based Reasoning with Extended Conceptual Logic Programs. In *Proc. of ESWC 2005*, number 3532 in LNCS, pages 392–407. Springer, 2005.
18. D. Kozen. Results on the Propositional μ-calculus. *Theor. Comput. Sci.*, 27:333–354, 1983.
19. J. Lee and V. Lifschitz. Loop Formulas for Disjunctive Logic Programs. In *Proc. of ICLP 2003*, volume 2916 of *LNCS*, pages 451–465. Springer, 2003.
20. V. Lifschitz, D. Pearce, and A. Valverde. Strongly Equivalent Logic Programs. *ACM Transactions on Computational Logic*, 2(4):526–541, 2001.
21. F. Lin and Y. Zhao. ASSAT: Computing Answer Sets of a Logic Program by SAT Solvers. In *Proc. of 18th National Conference on Artificial Intelligence*, pages 112–117. AAAI, 2002.
22. T. Syrjänen. Omega-restricted Logic Programs. In *Proc. of LPNMR*, volume 2173 of *LNAI*, pages 267–279. Springer, 2001.
23. T. Syrjänen and I. Niemelä. The SMODELS System. In *Proc. of LPNMR*, volume 2173 of *LNCS*, pages 434–438. Springer, 2001.

External Sources of Computation for Answer Set Solvers*

Francesco Calimeri and Giovambattista Ianni

Dipartimento di Matematica, Università della Calabria,
I-87036 Rende (CS), Italy
{calimeri, ianni}@mat.unical.it

Abstract. The paper introduces Answer Set Programming with External Predicates (ASP-EX), a framework aimed at enabling ASP to deal with external sources of computation. This feature is realized by the introduction of "parametric" *external predicates*, whose extension is not specified by means of a logic program but computed through external code. With respect to existing approaches it is explicitly addressed the issue of invention of new information coming from external predicates, in form of new, and possibly infinite, constant symbols. Several decidable restrictions of the language are identified as well as suitable algorithms for evaluating Answer Set Programs with external predicates. The framework paves the way to Answer Set Programming in several directions such as pattern manipulation applications, as well as the possibility to exploit function symbols. ASP-EX has been successfully implemented in the DLV system, which is now enabled to make external program calls.

1 Introduction

Among nonmonotonic semantics, Answer Set Programming (ASP) is nowadays taking a preeminent role, witnessed by the availability of efficient answer-set solvers, like ASSAT [Lin and Zhao, 2002], Cmodels [Babovich and Maratea, 2003], DLV [Leone *et al.*, 2005b], and Smodels [Simons *et al.*, 2002], and various extensions of the basic language with features such as classical negation, weak constraints, aggregates, cardinality and weight constraints. ASP has become an important knowledge representation formalism for declaratively solving AI problems in areas including planning [Eiter *et al.*, 2003], diagnosis and information integration [Leone *et al.*, 2005a], and more.

Despite these good results, state-of-the-art ASP systems hardly deal with data types such as strings, natural and real numbers. Although simple, this data types bring two kinds of technical problems: first, they range over infinite domains; second, they need to be manipulated with primitive constructs which can be encoded in logic programming at the cost of compromising efficiency and declarativity.Furthermore, interoperability with other software is nowadays

* Work supported by the EC under projects INFOMIX (IST-2001-3357) and WASP (IST-2001-37004), and by FWF under project "Answer Set Programming for the Semantic Web" (P17212-N04)

C. Baral et al. (Eds.): LPNMR 2005, LNAI 3662, pp. 105–118, 2005.
© Springer-Verlag Berlin Heidelberg 2005

important, especially in the context of those Semantic Web applications aimed at managing external knowledge.

The contributions of the paper are the following:

- we introduce a formal framework, named `ASP-EX`, for accommodating *external predicates* in the context of Answer Set Programming;
- `ASP-EX` includes the explicit possibility of invention of new values from external sources: since this setting could lead to non-termination of any conceivable evaluation algorithm, we tailor specific cases where decidability is preserved.
- we show that `ASP-EX` enhances the applicability of Answer Set Programming to a variety of problems such as string and algebraic manipulation. Also the framework paves the way for simulating function symbols in a setting where the notion of term is kept simple (Skolem terms are not necessary).
- we discuss implementation issues, and show how we have integrated `ASP-EX` in the `DLV` system, which is, this way, enabled with the possibility of using external sources of computation.
- we carry out some experiments, confirming that the accommodation of external predicates does not cause any relevant computational overhead.

2 Motivating Example

The introduction of external sources of computation in tight synergy with Answer Set Solvers opens a variety of possible applications. We show next an example of these successful experiences.

The discovery of complex pattern repetitions in string databases plays an important role in genomic studies, and in general in the areas of knowledge discovery. Genome databases mainly consist of sets of strings representing DNA or protein sequences (biosequences) and most of these strings still require to be interpreted. In this context, discovering common patterns in sets of biologically related sequences is very important.

It turns out that specifying pattern search strategies by means of Answer Set Programming and its extensions is an appealing idea: constructs like strong and weak constraints, disjunction, aggregates may help an algorithm designer to fast prototype search algorithms for a variety of pattern classes.

Unfortunately, state-of-the-art Answer Set Solvers lack the possibility to deal in a satisfactory way with infinite domains such as strings or natural numbers. Furthermore, although very simple, such data types need of ad hoc manipulation constructs, which are typically difficult to be encoded and cannot be efficiently evaluated in logic programming.

So, in order to cope with these needs, one may conceive to properly extend answer set programming with the possibility of introducing *external* predicates. The extension of an external predicate can be efficiently computed by means of an intensional definition expressed using a traditional imperative language.

Thus, we might allow a pattern search algorithm designer to take advantage of Answer Set Programming facilities, but extended with special atoms such as

e.g. *#inverse(S1,S2)* (true if *S1* is the inverse string of *S2*), *#strcat(S1,S2,S3)* (true if *S3* is equal to the concatenation of *S1* and *S2*), or *#hammingDistance(S1,S2,N)* (true if *S1* has N differences with respect to *S2*). Note that it is desirable that these predicates introduce new values in the domain of a program whenever necessary. For instance, the semantics of *#strcat(a,b,X)* should be such that X matches with the new symbol *ab*.

Provided with a suitable mechanism for defining external predicates, the authors of [Palopoli *et al.*, 2005] have been able to define and implement a framework allowing to specify and resolve genomic pattern search problems; the framework is based on automatically generating logic programs starting from user-defined extraction problems, and exploits disjunctive logic programming properly extended in order to enable the possibility of dealing with a large variety of pattern problems. The external built-in framework implemented into the `DLV` system is essential in order to deal with strings and patterns. We provide next syntax and semantics of the proposed framework.

3 Syntax and Semantics

Let $\mathcal{U}$, $\mathcal{X}$, $\mathcal{E}$ and $\mathcal{P}$ be mutually disjoint sets whose elements are called *constant names*, *variable names*, *external predicate names*, and *ordinary predicate names*, respectively. Unless explicitly specified, elements from $\mathcal{X}$ (resp., $\mathcal{U}$) are denoted with first letter in upper case (resp., lower case); elements from $\mathcal{E}$ are usually prefixed with "#". $\mathcal{U}$ will constitute the default *Herbrand Universe*. We will assume that any constant appearing in a program or generated by external computation is taken from $\mathcal{U}$, which is possibly infinite[1].

Elements from $\mathcal{U} \cup \mathcal{X}$ are called *terms*. An *atom* is a structure $p(t_1, \ldots, t_n)$, where $t_1, \ldots, t_n$ are terms and $p \in \mathcal{P} \cup \mathcal{E}$; $n \geq 0$ is the *arity* of the atom. Intuitively, p is the predicate name. The atom is *ordinary*, if $p \in \mathcal{P}$, otherwise we call it *external atom*. A list of terms $t_1, \ldots, t_n$ is succinctly represented by $\bar{t}$. A positive *literal* is an atom, whereas a *negative literal* is $\texttt{not}\, a$ where a is an atom.

For example, *node(X)*, and *#succ(a, Y)* are atoms; the first is ordinary, whereas the second is an external atom.

A *rule* r is of the form

$$\alpha_1 \vee \cdots \vee \alpha_k \leftarrow \beta_1, \ldots, \beta_n, \texttt{not}\, \beta_{n+1}, \ldots, \texttt{not}\, \beta_m, \tag{1}$$

where $m \geq 0$, $k \geq 1$, $\alpha_1, \ldots, \alpha_k$, are ordinary atoms, and $\beta_1, \ldots, \beta_m$ are (ordinary or external) atoms. We define $H(r) = \{\alpha_1, \ldots, \alpha_k\}$ and $B(r) = B^+(r) \cup B^-(r)$, where $B^+(r) = \{\beta_1, \ldots, \beta_n\}$ and $B^-(r) = \{\beta_{n+1}, \ldots, \beta_m\}$. $E(r)$ is the set of external atoms of r. If $H(r) = \emptyset$ and $B(r) \neq \emptyset$, then r is a *constraint*, and if $B(r) = \emptyset$ and $H(r) \neq \emptyset$, then r is a *fact*; r is *ordinary*, if it contains only ordinary atoms. A ***ASP-EX*** *program* is a finite set P of rules; it is *ordinary*, if all rules are

[1] Also, we assume that constants are encoded using some finite alphabet Σ, i.e. they are finite elements of Σ^*.

ordinary. Without loss of generality, we will assume P has no constraints[2] and only ground facts.

The dependency graph $G(P)$ of P is built in the standard way by inserting a node n_p for each predicate name p appearing in P and a directed edge (p_1, p_2), labelled r, for each rule r such that $p_2 \in B(r)$ and $p_1 \in H(r)$.

The following is a short ASP-EX program:

$$\begin{aligned} \mathit{mustChangePasswd(Usr)} \leftarrow\ & \mathit{passwd(Usr,Pass)}, \\ & \mathit{\#strlen(Pass,Len)}, \#< \mathit{(Len,8)}. \end{aligned} \tag{2}$$

We define the semantics of ASP-EX by generalizing the answer-set semantics, proposed by Gelfond and Lifschitz [1991] as an extension of the stable model semantics of normal logic programs [Gelfond and Lifschitz, 1988]. In the sequel, we will assume P is a ASP-EX program. The *Herbrand base* of P with respect to $\mathcal{U}$, denoted $HB_{\mathcal{U}}(P)$, is the set of all possible ground versions of ordinary atoms and external atoms occurring in P obtained by replacing variables with constants from $\mathcal{U}$. The grounding of a rule r, $grnd_{\mathcal{U}}(r)$, is defined accordingly, and the grounding of program P by $grnd_{\mathcal{U}}(P) = \bigcup_{r \in P} grnd_{\mathcal{U}}(r)$.

An *interpretation* I for P is a couple $\langle S, F \rangle$ where:

- $S \subseteq HB_{\mathcal{U}}(P)$ contains only ordinary atoms; We say that I (or by small abuse of notation, S) is a *model* of ordinary atom $a \in HB_{\mathcal{U}}(P)$, denoted $I \models a$ ($S \models a$), if $a \in S$.
- F is a mapping associating with every external predicate name $\#e \in \mathcal{E}$, a decidable n-ary Boolean function (which we will call *oracle*) $F(\#e)$ assigning each tuple $(x_1, \ldots, x_n)$ either 0 or 1, where n is the fixed arity of $\#e$, and $x_i \in \mathcal{U}$. I (or by small abuse of notation, F) is a *model* of a ground external atom $a = \#e(x_1, \ldots, x_n)$, denoted $I \models a$ ($F \models a$), if $F(\#e)(x_1, \ldots, x_n) = 1$.

A positive literal is modeled if its atom is modeled, whereas a negated literal is modeled if its corresponding atom is not modeled.

Example 1. We give an interpretation $I = \langle S, F \rangle$ such that the external predicate *#strlen* is associated to the oracle $F(\mathit{\#strlen})$, and $F(\#<)$ to $\#<$. Intuitively these oracles are defined such that *#strlen(pat4dat,7)* and $\#<$ *(7,8)* are modeled by I, whereas *#strlen(mypet,8)* and $\#<$ *(10,8)* are not.

The following is a ground version of rule 2:

$$\begin{aligned} \mathit{mustChangePasswd(frank)} \leftarrow\ & \mathit{passwd(frank,pat4dat)}, \\ & \mathit{\#strlen(pat4dat,7)}, \#< \mathit{(7,8)}. \end{aligned} \tag{3}$$

□

Let r be a ground rule. We define

[2] A constraint $\leftarrow B(r)$ can be easily simulated through the introduction of a corresponding standard rule *fail* ← *B(r),* `not` *fail*, where *fail* is a fresh predicate not occurring elsewhere in the program.

i. $I \models H(r)$ iff there is some $a \in H(r)$ such that $I \models a$;
ii. $I \models B(r)$ iff $I \models a$ for each atom $a \in B^+(r)$ and $I \not\models a$ for each atom $a \in B^-(r)$;
iii. $I \models r$ (i.e., r is satisfied) iff $I \models H(r)$ whenever $I \models B(r)$.

We say that I is a *model* of a ASP-EX program P with respect to a universe $\mathcal{U}$, denoted $I \models_{\mathcal{U}} P$, iff $I \models r$ for all $r \in grnd_{\mathcal{U}}(P)$. A model M is minimal if there is no model N such that $N \subset M$.

Given a general ground program P, its *GL reduct* w.r.t. an interpretation I is the positive ground program P^I, obtained from P by:

- deleting all rules having a negated literal which is not modeled by I;
- deleting all the negated literals from the remaining rules.

$I \subseteq HB_{\mathcal{U}}(P)$ is an answer set for a program P w.r.t. $\mathcal{U}$ iff I is a minimal model for the positive program $grnd_{\mathcal{U}}(P)^I$. Let $ans_{\mathcal{U}}(P)$ be the set of answer sets of $grnd_{\mathcal{U}}(P)$. We call P *F-satisfiable*, if it has some answer set for a fixed function mapping F, i.e. if there is some interpretation $\langle S, F \rangle$ which is an answer set. In the following we will assume the semantics associated to each external predicate is defined a priori, i.e. F is fixed.

4 Properties of ASP-EX Programs

Although simple in its definition, the above semantics does not give any hint on how to actually compute answer sets of a given program P. In general, given an infinite domain of constants $\mathcal{U}$, and a program P, $HB_{\mathcal{U}}(P)$ is indeed infinite.

Theorem 1. *It is given a ASP-EX program P, a domain of constants $\mathcal{U}$, and a function mapping F where the co-domain of F contains only boolean functions decidable in polynomial time in the size of their arguments. Deciding whether P is F-satisfiable in the domain $\mathcal{U}$ is undecidable.*

Proof. (Sketch) The proof is carried out by showing that the Answer Set Semantics of a ordinary program $\overline{P}$ with function symbols[3] can be reduced to the Answer Set Semantics of a ASP-EX program P. We take advantage of a family of external predicates $\{\#function_i\}$. In a given interpretation $\langle S, F \rangle$, F will be such that $\#function_i(C, f, x_1, \ldots, x_i)$ is modeled if C unifies with the compound term $f(x_1, \ldots, x_i)$.

This allows to rewrite a logic program $\overline{P}$ with function symbols by means of external predicates. For instance, given the rule

$$p(s(X)) \leftarrow a(X, f(Y, h(Z))).$$

This can be rewritten in an equivalent ASP-EX rule:

$$p(S) \leftarrow a(X, F), \quad \#function_1(S, s, X), \quad \#function_2(F, f, Y, H), \quad \#function_1(H, h, Z).$$

□

[3] Positive Horn programs with function symbols are undecidable, see e.g. [Dantsin *et al.*, 2001].

Tailoring cases where a finite portion of $\mathcal{U}$ is enough to evaluate the semantics of a given program is thus of interest. In the following we reformulate some results regarding splitting sets [Lifschitz and Turner, 1994].

Definition 1. Given a `ASP-EX` program P, a *splitting set* is a set of atoms $A \in HB_{\mathcal{U}}(P)$ such that for each atom $a \in A$, if $a \in H(r)$ for some $r \in grnd_{\mathcal{U}}(P)$, then $B(r) \cup H(r) \subseteq A$. The *bottom* $b_A(P)$ is the set of rules $\{r \mid r \in grnd_{\mathcal{U}}(P)$ and $H(r) \subseteq A\}$. The *residual* $r_{\mathcal{U}}(P, I)$ is a program obtained from $grnd_{\mathcal{U}}(P)$ by deleting all the rules which are not modeled by I, and removing from the remaining rules all the $a \in A$ modeled by I. □

We take advantage here of the formulation of the splitting theorem as given in [Bonatti, 2004].

Theorem 2. *(Splitting theorem [Lifschitz and Turner, 1994; Bonatti, 2004]) Given a program P and a splitting set A, $M \in ans_{\mathcal{U}}(P)$ iff M can be split in two disjoint sets I and J, such that $I \in ans_{\mathcal{U}}(b_A(P))$ and $J \in ans_{\mathcal{U}}(r_{\mathcal{U}}(grnd_{\mathcal{U}}(P) \setminus b_A(P)), I)$.*

Definition 2. Given a rule r, a variable X is *safe* in r if it appears in some ordinary atom $a \in B^+(r)$. A rule r is *safe* if each variable X appearing in r is safe. A program P is *safe* if each rule $r \in P$ is safe.

Theorem 3. *Given a safe* `ASP-EX` *program P, let $U \subset \mathcal{U}$ be the set of constants appearing in P. Then $ans_U(P) = ans_{\mathcal{U}}(P)$.*

Proof. (Sketch) The line of reasoning of the theorem is proving that, assuming P is safe, $grnd_U(P)$ is a finite splitting set for P. Furthermore, $grnd_U(P) = b_U(P)$. For each $M \in ans_U(P)$, we can prove that $r_{\mathcal{U}}(grnd_{\mathcal{U}}(P) \setminus b_U(P), M)$ is consistent and its only answer set is the empty model. Thus $M \cup \emptyset \in ans_{\mathcal{U}}(P)$. Viceversa, assuming an answer set $M \in ans_{\mathcal{U}}(P)$ is given, same arguments lead to conclude that $M \in ans_U(P)$. □

In case a safe program is given, the above theorem allows to consider as the set of "relevant" constants only those values explicitly appearing in the program at hand. Intuitively, the semantics of a safe program P can be evaluated by means of the following steps:

- compute $grnd_U(P)$;
- remove from $grnd_U(P)$ all the rules containing at least one external literal e such that $F \not\models e$, and remove from each rule all the remaining external literals.
- compute the remaining ordinary program by means of a standard Answer Set solver.

It is worth pointing out that, assuming the complexity of computing oracles is polynomial in the size of their arguments, this algorithm as same complexity as computing $grnd_U(P)$[4].

5 Dealing with Values Invention

Although important for clarifying the given semantics, it is an actual practice to specify external sources of computation not in terms of boolean oracles. So we aim at introducing the possibility to specify *functional oracles*, keeping anyway the simple reference semantics given previously. In the new setting we are going to introduce, it is also very important that an external atom brings knowledge from external sources of computation, in terms of new symbols added to a given program.

For instance, assume $\mathcal{U}$ contains encoded values that can be interpreted as natural numbers and that the external predicate *#sqr* is defined such that the atom *#sqr(X, Y)* is true whenever Y encodes a natural number representing the square of the natural number X; we want to extract a series of squared values from this predicate; consider the short program

$$\begin{array}{r}number(2) \leftarrow \\ square(Y) \leftarrow number(X), \#sqr(X,Y).\end{array} \quad (4)$$

In the presence of unsafe rules as in the above example, Theorem 3 ceases to hold: it is indeed unclear whether there is a finite set of constants which the program can be grounded on. In the above example, we can intuitively conclude that the set of meaningful constants is $\{2, 4\}$. It is however undecidable, given a computable boolean oracle f to establish whether a given set S contains all and only all those tuples $\bar{t}$ such that $f(\bar{t}) = 1$.

In order to overcome these limits, we extend our framework with the possibility of explicitly computing missing values on demand. Although restrictive, this setting is not far from a realistic scenario where external predicates are defined by means of generic partial functions instead of boolean ones.

Definition 3. It is given an external predicate name $\#p$, having arity n and its oracle function $F(\#p)$. A *pattern* is a list of b's and u's. A b will represent a placeholder for a constant (or a bounded variable), whereas an u will be a placeholder for a variable. Given a list of terms, the corresponding pattern will be given by replacing each constant with a b, and each variable with a u. □

For instance, the pattern related to the list of terms (X, a, Y) is (u, b, u). Let *pat* be a pattern of length n having k placeholders b (which we will call input positions), and $n-k$ placeholders of u type (which we will call output positions). A *functional oracle* $F(\#p)[pat]$ for the pattern *pat*, associated to the external

[4] Assuming rules can have unbounded length, grounding a disjunctive logic program is in the worst case exponential in the size of the Herbrand base (see e.g. [Leone *et al.*, 2001]).

predicate $\#p$, is a partial function taking k constant arguments from $\mathcal{U}$ and returning a tuple of arity $n-k$, and such that $F(\#p)[pat](a_1, ..., a_k) = b_1, ..., b_{n-k}$ iff $F(\#p)(a_1, \ldots, a_k, b_1, \ldots, b_{n-k}) = 1$. Let $pat[j]$ be the j-th element of a pattern pat. Let $unbound_{pat}(\overline{X})$ be the sub-list of $\overline{X}$ such that $pat[j] = u$ for each $X_j \in \overline{X}$, and $bound_{pat}(\overline{X})$ be the sub-list of $\overline{X}$ such that $pat[j] = b$ for each $X_j \in \overline{X}$.

An external predicate $\#p$ might be associated to one or more functional oracles "consistent" with the originating boolean oracle. For instance, consider the $\#sqr$ external predicate, defined as mentioned above. We associate to it two functional oracles, $F(\#sqr)[b, u]$ and $F(\#sqr)[u, b]$. The two functional oracles are such that, e.g.

$$F(\#sqr)[b, u](3) = 9 \tag{5}$$

$$F(\#sqr)[u, b](16) = 4 \tag{6}$$

consistently with the fact that $F(\#sqr)(3, 9) = F(\#sqr)(4, 16) = 1$, whereas $F(\#sqr)[u, b](5)$ is set as undefined since $F(\#sqr)(X, 5) = 0$ for any natural X.

In the sequel, given an external predicate $\#e$, we will assume it comes equipped with its oracle $F(\#e)$ (called also *base oracle*) and a list of consistent functional oracles $\{F(\#e)[pat_1], \ldots, F(\#e)[pat_m]\}$, having different patterns $pat_1, \ldots, pat_m$[5].

Adopting functional oracles in the context of safe programs is however to big a restriction. We thus aim at enlarging the class of programs that can be evaluated against a finite Herbrand universe. To this end, we introduce a relaxed notion of safety. Intuitively, a variable is weakly safe if its value, although not explicitly appearing in a program, can be computed through a functional oracle.

Definition 4. Given a rule r, let $E(r)$ its set of external atoms. A *choice C of functional oracles* is a mapping $C : E(r) \mapsto \mathbf{N}$ associating each external atom of r with the index of one of its functional oracles. Given a choice C, let $F_C(\#e)$ a shortcut for the functional oracle $F(\#e)[pat_{C(\#e)}]$.

Given a rule r and a choice C, a variable X is *weakly safe in r w.r.t. to C* if either

- X is safe; or
- X appears in some external atom $\#e(\overline{X}) \in B^+(r)$, $X \in unbound_{pat_{C(\#e)}}$ and each variable $Y \in bound_{pat_{C(\#e)}}$ is weakly safe. □

A rule r is weakly safe if there is a choice C_r such that each variable X appearing in some atom $a \in B(r)$ is weakly safe with respect to C_r. A program P is weakly safe if each rule $r \in P$ is weakly safe. □

Example 2. Assume that $\#sqr$ is associated to the list of functional oracles $\{F(\#sqr)[b, u],\ F(\#sqr)[u, b]\}$ defined above. Given a choice of oracles C such

[5] Note that functional oracles prevent, to some extent, to define multivalued functions and/or generic relations. We consider anyway this setting acceptable for a variety of applications.

that $C(\#sqr(X,Y)) = 2$, the second rule of Program 4 is not weakly safe (intuitively there is no way for computing the value of the variable Y with the oracle $F(\#sqr)[u,b]$. The same rule is weakly safe if we set $C(\#sqr(X,Y)) = 1$. □

It turns out that deciding whether a given rule is weakly safe or not depends on a given choice, but also from the set of available functional oracles. It is assumed indeed that an external predicate does not come with all its possible functional oracles.

Proposition 1. *Given a set of external predicates $\mathcal{E}$, and a list of functional oracles for each $\#\mathrm{e} \in \mathcal{E}$, it can be checked in polynomial time whether a program P is weakly safe.*

Proof. (Sketch) Simply observe that for each rule $r \in P$ it can be checked in time linear in the number of atoms of r whether a choice making r weakly safe exists. □

Weakly safe rules can be grounded with respect to functional oracles as follows.

Definition 5. Given a weakly safe rule r, a choice C for it, and a set of ordinary ground atoms A, a ground rule r' is member of $ins(r, A)$ if r can be grounded to r' by the following algorithm:

1. replace positive literals of r with a consistent nondeterministic choice of matching ground atoms from A; let θ the resulting variable substitution;
2. until θ instantiates all the variables of r:
   - pick from $r\theta$ an external atom $\#e(\overline{X})\theta$ such that θ instantiates all the variables $X \in bound_{pat_{C(\#e)}}(\overline{X})$.
   - If $F_C(\#e)(bound_{pat_{C(\#e)}}(\overline{X}\theta)) = a_1, \ldots, a_k$, then update θ by assigning $a_1, \ldots, a_k$ to $unbound_{pat_{C(\#e)}}(\overline{X}\theta)$; else fail;
3. return $r' = r\theta$. □

Example 3. *Let's consider the second rule of Program 4; then, $ins(r, \{number(1), number(2)\})$ contains the two rules:*

$$\begin{array}{l} square(1) \leftarrow number(1), \#sqr(1,1). \\ square(4) \leftarrow number(2), \#sqr(2,4). \end{array}$$

□

Although desirable, weak safety is not sufficient in order to intuitively guarantee finiteness of answer sets and decidability. For instance, the program

$$\begin{array}{r} square(2) \leftarrow \\ square(Y) \leftarrow square(X), \#sqr(X,Y). \end{array} \tag{7}$$

is modeled by the infinite set of atoms $\{square(2), square(4), \ldots\}$.

We thus introduce the notion of *semi-safe* program. Intuitively a semi-safe program is such that external atoms cannot create infinite chains of new values to be taken in account.

Definition 6. A weakly safe program P is *semi-safe* if each cycle in $G(P)$ contains only edges corresponding to safe rules. □

Example 4. *For instance, the program*

$$square(Y) \leftarrow square(X), number(Y), \#sqr(X,Y).$$
$$square(Y) \leftarrow number(X), \#sqr(X,Y).$$

is semi-safe. □

We extend next Theorem 3 to the case of semi-safe programs.

Theorem 4. *It is given a semi-safe program P. Then there is a finite set of constants U such that $ans_U(P) = ans_{\mathcal{U}}(P)$.*

Proof. (Sketch) The set U is defined as all the constant symbols appearing in the set of atoms $T_P^\infty(\emptyset)$ where the operator T_P is defined as follows.

$$T_P(A) = A \cup \{a \in H(r') | r' \in ins(r, A) \text{ for some } r \in P\}$$

It is provable that $T_P^\infty(\emptyset) = T_P^n(\emptyset)$ for some n in case P is semi-safe; $T_P^\infty(\emptyset)$ is a splitting set, and U is finite; as in Theorem 3 for each $M \in ans_U(P)$, we can prove that $r_{\mathcal{U}}(grnd_{\mathcal{U}}(P) \setminus b_{T_P^\infty}(P), M)$ is consistent and its only answer set is the empty model. Thus $M \cup \emptyset \in ans_{\mathcal{U}}(P)$. Assuming an answer set $M \in ans_{\mathcal{U}}(P)$ is given, same arguments lead to conclude that $M \in ans_U(P)$. □

The above theorem allows to compute semantics of a semi-safe program P by means of a traditional answer set solver, following the steps:

- compute the ground program $T_P^\infty(\emptyset)$. This computation involves a number of evaluation of $ins(r, A)$ that trigger evaluation of functional oracles whenever needed;
- eliminate external literals as in the case of safe programs;
- evaluate the remaining ordinary program by means of a traditional solver;

We observe that, assuming F contains polynomial-time functional oracles, the complexity of the above algorithm is not greater than the complexity of computing grounding for an ordinary program.

6 Implementation and Experiments

The proposed language has been integrated into the ASP system `DLV` [Leone *et al.*, 2005b]. We called this prototype `DLV-EX`. From a practical point of view, the external atoms are dealt with in the following steps (see Figure 1):

1. at design time: a developer provides a library of external atoms, each of them associated with a set of functional oracles. Each functional oracle has a corresponding pattern. Although useful in practice, it is not compulsory to provide functional oracles other than the base oracle. However, the absence of specific functional oracles limits *de facto* the possibility to exploit an external atom in weakly safe rules. A testing environment helps checking the correctness of the oracles by means of automatically generated test programs.

2. at run-time in a pre-processing stage: each rule is checked to be weakly safe, and a suitable choice of functional oracles is made. Then the overall program is checked to be semi-safe. It is anyway possible to relax this second condition, provided that termination of grounding algorithms is not guaranteed in this case. It is worth pointing out that an user developing a logic program is not in charge of specifying a choice of oracles, since the system itself will choose the best functional oracles among a variety of possibilities.
3. at run-time during the rule instantiation stage: the optimized grounder of the DLV system has been extended in order to compute $ins(r, A)$ for a given rule r and a set of "active" atoms A. For each external atom in r,the chosen functional oracles are repeatedly invoked according to Definition 5.

Point 2 and 3 above are integrated in the existing grounding algorithm of the system. We briefly recall the rule instantiation algorithm of the DLV system [Leone *et al.*, 2001]. Given a rule r, this algorithm exploits an intelligent backtracking algorithm, where a given atom $a \in B(r)$ is picked at each stage and it is tried to be instantiated with respect to currently allowed values. The picking order is crucial in order to tailor the search space to the smallest extent: in principle, it is preferred to pick first those atoms whose estimated set of possible values is smaller.

The presence of external atoms impacts within such algorithm in a twofold way: for what point 2 above is concerned, given a rule r, among possible choices of functional oracles, our algorithm prefers those patterns whose number of unbounded variables is bigger. This intuitively allows to reduce the space of possible instantiations for a given external atom. For instance, given the atom *#sqr(X, Y)*, the choosing algorithm prefers, whenever possible, to choose the oracle with pattern (b, u) instead of the base oracle (which can be seen has having the pattern (b, b)), since this way it is searched only the space of values where Y is equal to the square of X. In the second case, an oracle with pattern (b, b) forces in principle to check all the possible couples of values for X and Y.

Point 3 impacts on the atom pick-up ordering strategy. For the same reasons above, it is preferred to pick up external atoms, with pattern having many unbounded variables, as earlier as possible. This strategy relies on the assumption, often true in practice, that the computation of a functional oracle is less time consuming than several computations of the corresponding base oracle.

All the pre-existing built-in atoms available in the DLV system (such as arithmetic and relational operators) have been rewritten using the new general framework. We carried out some experiment in order to appreciate the impact and the possible overhead of the new construct. Results are encouraging: grounding times are in most cases equivalent, and the slowdown reported in few cases is never above 6-7%.

External predicate definitions can be grouped in one or more libraries. Libraries have to be compiled such that they can be dynamically linked to the DLV-EX executable; oracles are written in the C++ language. A special directive inside DLV-EX programs tells the system which libraries have to be linked at run-

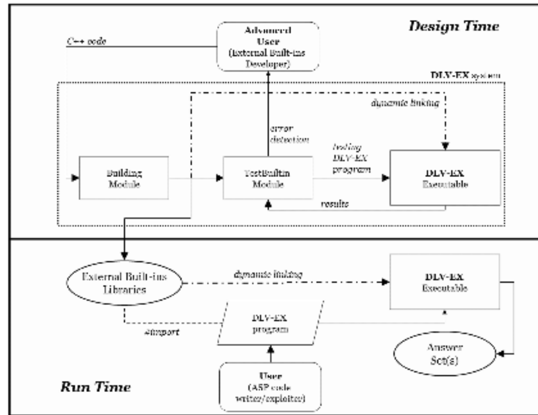


Fig. 1. System Architecture

time. Also, built-in developers are enabled to redefine predefined operators in order to deal with new data types, e.g. real numbers.

Some usage experiments have been carried out as well; few users have been requested to start implementing some customized libraries [Palopoli *et al.*, 2005; Cumbo *et al.*, 2004],and early feedbacks are positive both from the correctness and the ease of use points of view.

7 Related Works

For what the possibility of calling external modules in a logic program is concerned, it is worth to mention the foundational work of Eiter *et al.* [1997]. This paper takes the notion of generalized quantifier, known in formal logics, and adapts it in the context of modular logic programming. A generalized quantifier indeed, can be seen as a way for delegating the truth value of a formula to an external source of computation. Based on this work, the same authors are addressing the issue of implementing generalized quantifiers under Answer Set Semantics, in order to enable Answer Set Solvers to communicate, in both directions, with external reasoners [Eiter *et al.*, 2004; Eiter *et al.*, 2005]. This approach is different from the one considered in this paper since the former is inspired from second order logics and allows bidirectional flow of relational data (to and from an external atom), whereas, in our setting, the information flow is strictly value based. Nonetheless, HEX programs, as defined in [Eiter *et al.*, 2005], do not deal with infinite domains explicitly.

Although this know-how has not been explicitly divulgated yet, other Answer Set Solvers introduced the possibility to deal with externally computed functions [Syrjänen, 2002; Osorio and Corona, 2003].

Furthermore, there are several works aiming at bringing in Answer Set Programming a restricted capability of dealing with infinite domains. Among these, it is worth citing the notion of ω-restricted programs [Syrjänen, 2001]. ω -restricted programs allow to keep decidability of Answer Set Semantics in the presence of functions symbols, and constitute a subclass of finitary programs. It is indeed important to recall the work of Bonatti [2004], aimed at tailoring the class of finitary programs. Although, in general, recognizing this class of programs is undecidable, finitary programs allow function symbols but are decidable under brave/skeptical reasoning with ground queries. As shown in Theorem 1, external functions might be exploited in order to simulate function symbols. It is a matter of future search to extend the notion of semi-safe program to a larger class and investigate equivalence conditions with the notion of finitary program.

In the above cited literature, infinite domains are obtained through the introduction of compound functional terms. Thus the studied theoretical insights are often specialized to this notion of term, and take advantage e.g., of the common unification rules of formal logics over infinite domains. Similar in spirit to our approach is the work on open logic programs, and conceptual logic programs [Heymans *et al.*, 2004]. Such paper addresses the possibility of grounding a logic program, under Answer Set Semantics, over an infinite domain, in a way similar to classical logics and/or description logics. Each constant symbol has no predefined compound structure however. Also similar is the work of Cabibbo [1998], which extend the work of Hull and Yoshikawa [1998]. The latter authors introduce a language (ILOG) with a special construct aimed at introducing new invented values in a logic program, for the purpose of creating new tuple identifiers in relational databases. Based on this work, Cabibbo investigates about decidable fragments of the language. Despite some crucial semantic differences, the presented notion of weak safety is similar to the one herein presented, and describes conditions such that new values do not propagate in infinite chains.

8 Conclusions

We presented a framework where external atoms with value invention are taken in account. The purpose of this work is in the direction of closing the gap between Answer Set Programming and practical applications. Also, we believe this works paves the way to an actual implementation of finitary programs with function symbols. The system prototype, examples, manuals and benchmark results are available at `http://www.mat.unical.it/kali/dlv-ex`.

References

[Babovich and Maratea] Y. Babovich and M. Maratea. Cmodels-2: Sat-based answer sets solver enhanced to non-tight programs. http://www.cs.utexas.edu/users/tag/cmodels.html, 2003.

[Bonatti] P. A. Bonatti. Reasoning with infinite stable models. *AI*,156(1):75–111,2004.

[Cabibbo] L. Cabibbo. The Expressive Power of Stratified Logic Programs with Value Invention. *Inf. Comput.*, 147(1):22–56, 1998.

[Cumbo *et al.*] C. Cumbo, S. Iiritano, and P. Rullo. Reasoning-based knowledge extraction for text classification. In *Discovery Science*, pp. 380–387, 2004.

[Dantsin *et al.*] E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov. Complexity and Expressive Power of Logic Programming. *ACM CS*, 33(3):374–425, 2001.

[Eiter *et al.*] T. Eiter, G. Gottlob, and H. Veith. Modular Logic Programming and Generalized Quantifiers. In *LPNMR-97*, LNCS 1265.

[Eiter *et al.*] T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres. A Logic Programming Approach to Knowledge-State Planning, II: the DLV$^{\mathcal{K}}$ System. *Artif. Intell.*, 144(1–2):157–211, 2003.

[Eiter *et al.*] T. Eiter, G. Ianni, R. Schindlauer, and H. Tompits. Nonmonotonic description logic programs: Implementation and experiments. In *LPAR 2004*, pp. 511–527, 2004.

[Eiter *et al.*] T. Eiter, G. Ianni, R. Schindlauer, and H. Tompits. A Uniform Integration of Higher-Order Reasoning and External Evaluations in Answer Set Programming. In *IJCAI 2005*, to appear, Edinburgh, UK, 2005.

[Gelfond and Lifschitz] M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programming. In *ICLP 1988*, pp. 1070–1080, MIT Press.

[Gelfond and Lifschitz] M. Gelfond and V. Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *NGC*, 9:365–385, 1991.

[Heymans *et al.*] S. Heymans, D. Van Nieuwenborgh, and D. Vermeir. Semantic web reasoning with conceptual logic programs. In *RuleML 2004*, pp. 113–127, 2004.

[Hull and Yoshikawa] R. Hull and M. Yoshikawa. On the equivalence of database restructurings involving object identifiers. In *PODS 1991*, pp. 328–340. ACM Press.

[Leone *et al.*] N. Leone, S. Perri, and F. Scarcello. Improving ASP Instantiators by Join-Ordering Methods. *LPNMR'01, Vienna, Austria, 2001*, LNCS 2173.

[Leone *et al.*] N. Leone, G. Gottlob, R. Rosati, T. Eiter et al. The INFOMIX System for Advanced Integration of Incomplete and Inconsistent Data. In *SIGMOD 2005*, ACM, to appear.

[Leone *et al.*] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The DLV System for Knowledge Representation and Reasoning. *ACM TOCL*, 2005. To appear.

[Lifschitz and Turner] V. Lifschitz and H. Turner. Splitting a Logic Program. In *ICLP'94*, pp. 23–37, MIT Press.

[Lin and Zhao] F. Lin and Y. Zhao. ASSAT: Computing Answer Sets of a Logic Program by SAT Solvers. In *AAAI-2002*, AAAI Press / MIT Press.

[Osorio and Corona] M. Osorio and Enrique Corona. The A-Pol system. In *ASP*,2003.

[Palopoli *et al.*] L. Palopoli, S. Rombo, and G. Terracina. Flexible Pattern Discovery with (Extended) Disjunctive Logic Programming. In *International Symposium on Methodologies for Intelligent Systems (ISMIS 2005)*, pp. 504–513, Verlag.

[Simons *et al.*] P. Simons, I. Niemelä, and T. Soininen. Extending and Implementing the Stable Model Semantics. *Artif. Intell.*, 138:181–234, 2002.

[Syrjänen] T. Syrjänen. Omega-restricted logic programs. In *LPNMR-6*, Vienna, Austria, 2001. Verlag.

[Syrjänen] T. Syrjänen. Lparse 1.0 User's Manual, 2002.

Answer Sets for Propositional Theories

Paolo Ferraris

Department of Computer Sciences, The University of Texas at Austin,
Austin TX 78712, USA
otto@cs.utexas.edu

Abstract. Equilibrium logic, introduced by David Pearce, extends the concept of an answer set from logic programs to arbitrary sets of formulas. Logic programs correspond to the special case in which every formula is a "rule" — an implication that has no implications in the antecedent (body) and consequent (head). The semantics of equilibrium logic looks very different from the usual definitions of an answer set in logic programming, as it is based on Kripke models. In this paper we propose a new definition of equilibrium logic which uses the concept of a reduct, as in the standard definition of an answer set. Second, we apply the generalized concept of an answer set to the problem of defining the semantics of aggregates in answer set programming. We propose, in particular, a semantics for weight constraints that covers the problematic case of negative weights. Our semantics of aggregates is an extension of the approach due to Faber, Leone, and Pfeifer to a language with choice rules and, more generally, arbitrary rules with nested expressions.

1 Introduction

Equilibrium logic, introduced by Pearce [1997, 1999], extends the concept of an answer set [Gelfond and Lifschitz, 1988, 1991] from logic programs to arbitrary sets of formulas. Logic programs correspond to the special case in which every formula is a "rule" — an implication that has no implications (or equivalences) in the antecedent (body) and consequent (head).

The semantics of equilibrium logic looks very different from the usual definitions of an answer set in logic programming: it is based on Kripke models. In this paper, we propose a new definition of equilibrium logic, equivalent to Pearce's definition, which uses the concept of a reduct, as in the one used in the standard definition of an answer sets.

Second, we apply the generalized concept of an answer set to the problem of defining the semantics of aggregates in answer set programming. The best proposal in this area published so far is due to Faber, Leone, and Pfeifer [2004]. The strong point of that paper is that it is applicable to aggregates that are neither monotone nor antimonotone (such as, for instance, weight constraints in which some some weights are positive and some are negative). It has two defects, however. First, it does not allow negation in aggregate expressions. Second, it does not cover choice rules, which play an important role in answer set programming.[1]

[1] Both negation within aggregates and choice rules can be defined, in principle, as abbreviations for expressions containing auxiliary atoms.

C. Baral et al. (Eds.): LPNMR 2005, LNAI 3662, pp. 119–131, 2005.
© Springer-Verlag Berlin Heidelberg 2005

Our semantics includes aggregates in the style of [Faber *et al.*, 2004] containing arbitrary formulas and also choice rules. We show also that the existence of an answer set for nondisjunctive program with aggregates of a very simple kind (weight constraints with the weights 1 and -1) is Σ_2^P-hard, as in the case of disjunctive programs.

In the following section we define answer sets for propositional theories, and we relate this definition to equilibrium logic and to the traditional definition of an answer set. In Section 3 we extend several important theorems about logic programs to propostional theories. In Section 4, we propose our semantics of aggregates, discuss its properties, and show, as an example, how it applies to representing a combinatorial auction with negative costs. Comparisons with other formalizations are given in Section 5.

2 Formulas, Reducts and Answer Sets

2.1 Definition

For simplicity, we limit our attention to formulas without strong negation. We consider (propositional) formulas formed from atoms and connectives $\bot$, $\vee$, $\wedge$ and $\supset$.[2] A *theory* is a set of formulas. In the rest of the paper, F and G denote formulas, Γ a theory, X and Y sets of atoms, and $\otimes$ a binary connective.

We identify an interpretation with the set of atoms satisfied by it. We write $X \models F$ ($X \models \Gamma$) if X satisfies F (or Γ) in the sense of classical logic.

The *reduct* F^X of F relative to X is defined recursively:

- if $X \not\models F$ then $F^X = \bot$,
- if $X \models a$ (a is an atom) then $a^X = a$, and
- if $X \models F \otimes G$ then $(F \otimes G)^X = F^X \otimes G^X$.

This definition of a reduct is similar to a transformation proposed in [Osorio *et al.*, 2004, Section 4.2].

The reduct F^X can be alternatively defined as the formula obtained from F by replacing every outermost subformula not satisfied by X with $\bot$ (this alternative definition applies even if we treat $\neg$, $\top$ and $\equiv$ as primitive connectives).

For instance, if X contains p but not q then

$$((p \supset q) \vee (q \supset p))^X = \bot \vee (\bot \supset p).$$

It is easy to see that, for every X, Y, $\otimes$, F and G,

$$Y \models (F \otimes G)^X \text{ iff } X \models F \otimes G \text{ and } Y \models F^X \otimes G^X. \tag{1}$$

The *reduct* Γ^X of Γ relative to X is $\{F^X : F \in \Gamma\}$. A set X is an *answer set* for Γ if X is a minimal set satisfying Γ^X.

For instance, let Γ be $\{(p \supset q) \vee (q \supset p), p\}$. Set $\{p\}$ is an answer set for Γ because $\{p\}$ is a minimal model satisfying the reduct $\{\bot \vee (\bot \supset p), p\}$. It is not difficult to see that no other set of atoms is an answer set for Γ.

[2] $\neg F$ stands for $F \supset \bot$; $\top$ stands for $\bot \supset \bot$; $F \equiv G$ stands for $(F \supset G) \wedge (G \supset F)$.

2.2 Relationship to Equilibrium Logic

Theorem 1. *For any theory, its models in the sense of equilibrium logic are identical to its answer sets.*

Since in application to programs with nested expressions equilibrium logic is equivalent to the semantics defined in [Lifschitz *et al.*, 1999], Theorem 1 implies that our definition of an answer set extends the corresponding definition from that paper.

In the proof of Theorem 1, we write $\langle X, Y \rangle \models \Gamma$ (with $X \subseteq Y$) if the HT-interpretation [3] $\langle X, Y \rangle$ is a model of Γ.

Lemma 1. *For any X and Y such that $X \subseteq Y$ and any theory Γ,*

$$X \models \Gamma^Y \textit{ iff } \langle X, Y \rangle \models \Gamma.$$

The lemma is proven first for the case when Γ is a singleton, by structural induction.

Proof of Theorem 1. According to the semantics of equilibrium logic (reproduced in [Lifschitz *et al.*, 2001, Section 4.4]), Y is a model of Γ iff

$$\langle Y, Y \rangle \models \Gamma \text{ and, for all proper subsets } X \text{ of } Y,\ \langle X, Y \rangle \not\models \Gamma.$$

In view of Lemma 1, this is the same as

$$Y \models \Gamma^Y \text{ and, for all proper subsets } X \text{ of } Y,\ X \not\models \Gamma^Y.$$

which means that Y is an answer set for Γ. □

2.3 Relationship to the Traditional Definition of Reduct

A *nested expressions* is a formula that contains no implications $F \supset G$ with $G \neq \bot$, and no equivalences. [4] A *program with nested expressions* is a set of *rules* $F \leftarrow G$, where F and G are nested expressions. We will identify such a rule with the implication $G \supset F$.

In application to programs with nested expressions, our definition of a reduct is quite different from the traditional definition [Lifschitz *et al.*, 1999]. Consider, for instance, the following program:

$$\begin{array}{l} p \leftarrow not\ q \\ q \leftarrow not\ r \end{array}$$

According to [Lifschitz *et al.*, 1999], its reduct relative to $\{r\}$ is

$$\begin{array}{l} p \leftarrow \top \\ q \leftarrow \bot; \end{array}$$

with our definition, it is

[3] See [Lifschitz *et al.*, 2001, Section 2.1].

[4] Traditionally, in nested expressions conjunction is denoted by comma, disjunction by semicolon, and negation by "*not*".

$$\bot$$
$$q \leftarrow \bot.$$

The first reduct is satisfied, for instance, by $\{p\}$, while the second is unsatisfiable. However, some similarities between these formalisms exist. For instance, it is easy to see that for any formula F, $(\neg F)^X$, according to the new definition, is $\top$ when $X \not\models F$, and $\bot$ otherwise, as with the traditional definition. Indeed, if $X \models F$ then $X \not\models F \supset \bot$ and consequently

$$(\neg F)^X = (F \supset \bot)^X = \bot.$$

Otherwise, $X \models F \supset \bot$, so that

$$(\neg F)^X = (F \supset \bot)^X = F^X \supset \bot = \bot \supset \bot = \top.$$

The following proposition states a more general relationship between the new definition of the reduct and the traditional one. We denote by $F^{\underline{X}}$ the reduct of a nested expression F relative to X according to the definition from [Lifschitz *et al.*, 1999], and similarly for the reduct of a program.

Proposition 1. *For any program Π with nested expressions and any set X of atoms, Π^X is equivalent, in the sense of classical logic,*

- *to $\bot$, if $X \not\models \Pi$, and*
- *to the program obtained from $\Pi^{\underline{X}}$ by replacing all atoms that do not belong to X by $\bot$, otherwise.*

The proof of this proposition is based on the following lemma, proven by structural induction.

Lemma 2. *The reduct F^X of a nested expression F is equivalent, in the sense of classical logic, to the nested expression obtained from $F^{\underline{X}}$ by replacing all atoms that do not belong to X by $\bot$.*

Corollary 1. *Given two sets of atoms X and Y with $Y \subseteq X$ and any program Π, $Y \models \Pi^X$ iff $X \models \Pi$ and $Y \models \Pi^{\underline{X}}$.*

This corollary suggests another way to verify that the definition of an answer set proposed in this paper is equivalent to the usual one in the case of programs with nested expressions. If $X \not\models \Pi$ then X is not an answer set for Π under either semantics. Otherwise, for every subset Y of X, $Y \models \Pi^X$ iff $Y \models \Pi^{\underline{X}}$ by Corollary 1.

3 Properties of Propositional Theories

Several theorems about answer sets for logic programs can be extended to propositional theories. The proofs are omitted for lack of space.

Two theories Γ_1 and Γ_2 are *strongly equivalent* if, for every theory Γ, $\Gamma_1 \cup \Gamma$ and $\Gamma_2 \cup \Gamma$ have the same answer sets.

Proposition 2. *For any two theories Γ_1 and Γ_2, the following conditions are equivalent:*

(i) Γ_1 *is strongly equivalent to* Γ_2*,*
(ii) Γ_1 *is equivalent to* Γ_2 *in the logic of here-and-there, and*
(iii) for each set X *of atoms,* Γ_1^X *is equivalent to* Γ_2^X *in classical logic.*

The equivalence between (i) and (ii) is a generalization of the main result of [Lifschitz *et al.*, 2001], and it is an immediate consequence of Lemma 4 from that paper and our Theorem 1. The equivalence between (i) and (iii) is similar to Theorem 1 from [Turner, 2003].

The following claims require some definitions. An occurrence of an atom in a formula is *positive* if it is in the antecedent of an even number of implications. An occurrence is *strictly positive* if such number is 0. An occurrence of an atom in a formula is *negated* if it is in a subformula of the form $F \supset \bot$. For instance, in a formula $(p \supset \bot) \supset q$, the occurrences of p and q are positive, the one of q is strictly positive, and the one of p is negated.

The following proposition is an extension of the property that in each answer set of a program, each atom occurs in the head of a rule of that program [Lifschitz, 1996, Section 3.1].

Proposition 3. *Each answer set of a theory consists of atoms that have a strictly positive occurrence in some formula of that theory.*

The following two propositions were stated in [Ferraris and Lifschitz, 2005] in the case of logic programs.

Proposition 4 (Lemma on Explicit Definitions). *Let* Γ *be any propositional theory, and* Q *a set of atoms that do not occur in* Γ*. For each* $q \in Q$*, let* $Def(q)$ *be a formula that doesn't contain any atom from* Q*. Then* $X \mapsto X \setminus Q$ *is a 1–1 correspondence between the answer sets of* $\Gamma \cup \{Def(q) \supset q : q \in Q\}$ *and the answer sets of* Γ*.*

Proposition 5 (Completion Lemma). *Let* Γ *be any propositional theory, and* Q *a set of atoms that do not have positive, nonnegated occurrences in any rule of* Γ*. For each* $q \in Q$*, let* $Def(q)$ *be a formula such that all occurrences of elements of* Q *in* $Def(q)$ *are either positive or negated. Then* $\Gamma \cup \{Def(q) \supset q : q \in Q\}$ *and* $\Gamma \cup \{Def(q) \equiv q : q \in Q\}$ *have the same answer sets.*

The following proposition is essentially a generalization of the splitting set theorem from [Lifschitz and Turner, 1994] and [Erdoğan and Lifschitz, 2004].

Proposition 6 (Splitting Set Theorem). *Let* Γ_1 *and* Γ_2 *be two theories such that all atoms occurring in* Γ_1 *have no strictly positive occurrences in* Γ_2*. Then a set of* X *is an answer set for* $\Gamma_1 \cup \Gamma_2$ *iff there is an answer set* Y *for* Γ_1 *such that* X *is an answer set for* $Y \cup \Gamma_2$*.*

4 Representing Aggregates

4.1 Definition

Aggregates are an important extension to logic programs, widely used in answer set programming. We define a *(ground) aggregate* as an expression of the form

$$op\langle\{F_1 = w_1, \dots, F_n = w_n\}\rangle \prec N \tag{2}$$

where

- op is (a symbol for) a function from multisets of $\mathcal{R}$ (real numbers) to $\mathcal{R} \cup \{-\infty, +\infty\}$ (such as sum, product, min, max, etc.),
- $\{F_1 = w_1, \dots, F_n = w_n\}$ $(n \geq 0)$ is a multiset where $F_1, \dots, F_n$ are formulas, and $w_1, \dots, w_n$ are (symbols for) real numbers ("weights"),
- $\prec$ is (a symbol for) a binary relation between real numbers, such as $\leq$ and $=$, and
- N is (a symbol for) a real number.

As an intuitive explanation of an aggregate, take the multiset W consisting of the weights w_i $(1 \leq i \leq n)$ such that F_i is "true". The aggregate is considered "true" if $op(W) \prec N$. For example,

$$sum\langle\{p = 1, q = 1\}\rangle \neq 1. \tag{3}$$

intuitively expresses the condition that both p are q are "true" or none of them.

To define the semantics of aggregates, we propose to identify (2) with the formula

$$\bigwedge_{I \subseteq \{1,\dots,n\} \;:\; op(\{w_i \;:\; i \in I\}) \not\prec N} \Big(\big(\bigwedge_{i \in I} F_i \big) \supset \big(\bigvee_{i \in \overline{I}} F_i \big) \Big), \tag{4}$$

where $\overline{I}$ stands for $\{1, \dots, n\} \setminus I$, and $\not\prec$ is the negation of $\prec$ [5].

For instance, if we consider aggregate (3), the conjunctive terms in (4) correspond to the cases when the sum of weights is 1, that is, when $I = \{1\}$ and $I = \{2\}$. The two implications are $q \supset p$ and $p \supset q$ respectively, so that (3) is

$$(q \supset p) \wedge (p \supset q). \tag{5}$$

Similarly,

$$sum\langle\{p = 1, q = 1\}\rangle = 1 \tag{6}$$

is

$$(p \vee q) \wedge \neg(p \wedge q). \tag{7}$$

Note that, even if (5) is classically equivalent to (7), they are not equivalent in the logic of here-and-there. This shows that it is generally incorrect to "move" a negation from a binary relation symbol (such as $\neq$) in front of the aggregate as the unary connective $\neg$.

Some properties of aggregates are stated in the following proposition.

[5] This definition, based on the idea of the translation from [Faber *et al.*, 2004], is meant to provide a very general semantics for aggregates, but we do not propose to use it directly for computing answer sets.

Proposition 7. *For any aggregate* $op\langle S\rangle \prec N$ *where* S *is*

$$\{F_1 = w_1, \dots, F_n = w_n\},$$

and any sets X *and* Y *of atoms,*

(a) $X \models op\langle S\rangle \prec N$ *iff* $op(\{w_i \ : \ X \models F_i\}) \prec N$*, and*
(b) $Y \models (op\langle S\rangle \prec N)^X$ *iff* $X \models op\langle S\rangle \prec N$ *and* $Y \models op\langle S^X\rangle \prec N$*,*

where S^X *stands for* $\{F_1^X = w_1, \dots, F_n^X = w_n\}$*.*

Proposition 7(a) confirms that our proposal to identify (2) with (4) is in agreement with the intuitive meaning of an aggregate. Part (b) is similar to property (1) of binary connectives.

When a theory Γ is described using abbreviation (2) in its formulas, answer sets of Γ can be computed using Proposition 7 instead of a direct reference to (4).

Finally, it can be shown by Proposition 2 that if we want to identify (2) with a formula so that Proposition 7 holds then (4) is the only choice, modulo strong equivalence.

Proposition 7(a) follows from the fact that X satisfies an implication in (4) iff $I \neq \{j \ : \ X \models F_j\}$. The proof of part (b) uses the following lemma that is easily provable.

Lemma 3. *For any formulas* $F_1, \dots, F_n$ $(n \geq 0)$*, any set* X *of atoms, and any connective* $\otimes \in \{\vee, \wedge\}$*,* $(F_1 \otimes \cdots \otimes F_n)^X$ *is classically equivalent to* $F_1^X \otimes \cdots \otimes F_n^X$*.*

4.2 Monotone Aggregates

An aggregate $op\langle\{F_1 = w_1, \dots, F_n = w_n\}\rangle \prec N$ is *monotone* if, for each pair of multisets W_1, W_2 such that $W_1 \subseteq W_2 \subseteq \{w_1, \dots, w_n\}$, $op(W_2) \prec N$ is true whenever $op(W_1) \prec N$ is true. The definition of an *antimonotone* aggregate is similar, with $W_1 \subseteq W_2$ replaced by $W_2 \subseteq W_1$.

For instance,

$$sum\langle\{p = 1, q = 1\}\rangle > 1. \tag{8}$$

is monotone, and

$$sum\langle\{p = 1, q = 1\}\rangle < 1. \tag{9}$$

is antimonotone. An example of an aggregate that is neither monotone nor antimonotone is (3).

Proposition 8. *An aggregate* $op\langle\{F_1 = w_1, \dots, F_n = w_n\}\rangle \prec N$ *is equivalent, in the logic of here-and-there, to*

$$\bigwedge_{I \subseteq \{1,\dots,n\} \ : \ op(\{w_i \ : \ i \in I\}) \not\prec N} \Big(\bigvee_{i \in \overline{I}} F_i\Big)$$

if the aggregate is monotone, and to

$$\bigwedge_{I \subseteq \{1,\dots,n\} \ : \ op(\{w_i \ : \ i \in I\}) \not\prec N} \Big(\neg \bigwedge_{i \in I} F_i\Big)$$

if the aggregate is antimonotone.

In other words, if $op\langle S\rangle \prec N$ is monotone then the antecedents of the implications in (4) can be dropped. Similarly, in case of antimonotone aggregates, the consequents of these implications can be replaced by $\bot$. In both cases, (4) is turned into a nested expression, if $F_1, \ldots, F_n$ are nested expressions.

For instance, the monotone aggregate (8) is

$$(p \vee q) \wedge (p \supset q) \wedge (q \supset p),$$

which is equivalent, in the logic of here and there, to

$$(p \vee q) \wedge q \wedge p$$

and then to $q \wedge p$. In the case of the antimonotone aggregate (9), the formula

$$((p \wedge q) \supset \bot) \wedge (p \supset q) \wedge (q \supset p)$$

is equivalent, in the logic of here-and-there, to

$$(\neg(p \wedge q)) \wedge \neg p \wedge \neg q,$$

and then to $\neg p \wedge \neg q$.

On the other hand, if an aggregate is neither monotone nor antimonotone, it may be not possible to find a nested expression equivalent, in the logic of here-and-there, to (4), even if $F_1, \ldots, F_n$ are nested expressions. This is the case for (3). Indeed, let A denote (3). Considering that this expression stands for (5), it is easy to check that $\langle\{p\}, \{p, q\}\rangle \not\models A$ and $\langle\emptyset, \{p, q\}\rangle \models A$. On the other hand, for any nested expression F, if $\langle\{p\}, \{p, q\}\rangle \not\models F$ then $\langle\emptyset, \{p, q\}\rangle \not\models F$ (easily provable by structural induction.)

Both parts of Proposition 8 can be proven, in the difficult direction, by strong induction on the cardinality of I.

4.3 Example

We consider the following variation of the combinatorial auction problem, which can be naturally formalized using an aggregate that is neither monotone nor antimonotone.

Joe wants to move to another town and has the problem of removing all his bulky furniture from his old place. He has received some bids: each bid may be for one piece or several pieces of furniture, and the amount offered can be negative (if the value of the pieces is lower than the cost of removing them). A junkyard will take any object not sold to bidders, for a price. The goal is to find a collection of bids for which Joe doesn't lose money, if there is any.

Assume that there are n bids, labeled from 1 to n. We express by the formulas

$$b_i \vee \neg b_i \tag{10}$$

($1 \leq i \leq n$) that Joe is free to accept any bid or not. Clearly, Joe cannot accept two bids that involve the selling of the same piece of furniture. So, for each pair i, j of such bids, we include the formula

$$\neg(b_i \wedge b_j). \tag{11}$$

Next, we need to express which pieces of the furniture have not been given to bidders. If there are m objects (numbered from 1 through m), we can express that an object i is sold by bid j by adding the rule

$$b_j \supset s_i \tag{12}$$

to our theory.

Finally, we need to express that Joe doesn't lose money by selling his items. This is done by the aggregate

$$sum\langle\{b_1 = w_1, \ldots, b_n = w_n, \neg s_1 = -c_1, \ldots, \neg s_m = -c_m\}\rangle \geq 0, \tag{13}$$

where each w_i is the amount of money (possibly negative) obtained by accepting bid i, and each c_i is the money requested by the junkyard to remove item i. Note that (13) is neither monotone nor antimonotone.

Proposition 9. *$X \mapsto X \cap \{b_1, \ldots, b_n\}$ is a 1–1 correspondence between the answer sets of the theory consisting of formulas (10)–(13) and the solutions of this problem.*

5 Other Formalisms

5.1 Programs with Weight Constraints

Weight constraints [Simons *et al.*, 2002] can be viewed as aggregates of the form $sum\langle S\rangle \geq N$ (traditionally denoted by $N \leq S$) and $sum\langle S\rangle \leq N$ (denoted by $S \leq N$), where each formula in S is a literal. Weight constraints are one of the most commonly used kind of aggregates in logic programs, especially in the case of weights that are equal to 1 (cardinality constraints).

A *program with weight constraints* is a set of formulas of the form

$$W_1 \wedge \cdots \wedge W_n \supset a \tag{14}$$

($n \geq 0$) where a is an atom or $\bot$, and $W_1, ..., W_n$ are weight constraints. [6]

Theorem 2. *For every program with weight constraints, if all the weights are positive, then the answer sets under our semantics are identical to its answer sets in the sense of [Simons* et al.*, 2002, Section 2.3].*

The proof consists in showing that, in the case of positive weights, $sum\langle S\rangle \geq N$ and $sum\langle S\rangle \leq N$ are equivalent, in the logic of here-and-there, to the nested expressions $[N \leq S]$ and $[S \leq N]$ defined in [Ferraris and Lifschitz, 2005]. Theorem 2 follows from this fact in view of Theorem 1 from [Ferraris and Lifschitz, 2005].

[6] For simplicity, we are considering only part of the syntax allowed in [Simons *et al.*, 2002]. Every rule in the sense of that paper can be equivalently rewritten as a set of rules of the form (14).

Our semantics is not equivalent to the semantics of [Simons *et al.*, 2002] when the weights can be negative (as discussed in the introduction, our view of negative weights is equivalent to the one proposed in [Faber *et al.*, 2004]). According to [Ferraris and Lifschitz, 2005, Footnote 6], the traditional semantics for weight constraints may lead to some unintuitive results: program

$$(0 \leq \{p = 2, p = -1\}) \supset p \tag{15}$$

according to [Simons *et al.*, 2002], has no answer sets, while

$$(0 \leq \{p = 1\}) \supset p$$

has one answer set $\{p\}$. Under our semantics, $\{p\}$ is the only answer set for both programs.

While weight constraints with positive weights only are either monotone or antimonotone, this is not the case when negative weights are allowed as in (13). In particular, it may not be possible to represent an aggregate of this kind by a nested expression.

This is the main reason why the translation from programs with weight constraints to programs with nested expressions of [Ferraris and Lifschitz, 2005] was limited to the case of positive weights only.

5.2 Complexity of Programs with Weight Constraints

Under the semantics of [Simons *et al.*, 2002], the existence of an answer set for programs with weight constraints is an NP-complete problem even in presence of negative weights. On the other hand, under our semantics, the place of this problem in the polynomial hierarchy is different.

Proposition 10. *Under the semantics of this paper, the existence of an answer set for a program with weight constraints is a Σ^2_P-complete problem.*

A similar result has been independently proven in [Calimeri *et al.*, 2005].

The problem is clearly in Σ^2_P by the definition of an answer set. The Σ^2_P-hardness follows from the Σ^2_P-completeness of the existence of an answer set for disjunctive logic programs [Eiter and Gottlob, 1993, Corollary 3.8], and the following lemma, which provides a polynomial translation from disjunctive programs to programs with weight constraints.

Lemma 4. *Rule*

$$l_1 \wedge \cdots \wedge l_m \supset a_1 \vee \cdots \vee a_n$$

($n > 0, m \geq 0$) where $a_1, \ldots, a_n$ are atoms and $l_1, \ldots, l_m$ are literals, is strongly equivalent to the set of n implications ($i = 1, \ldots, n$)

$$(1 \leq \{l_1 = 1\}) \wedge \cdots \wedge (1 \leq \{l_m = 1\}) \wedge A_{i1} \wedge \cdots \wedge A_{in} \supset a_i,$$

where each A_{ij} stands for $0 \leq \{a_i = 1, a_j = -1\}$.

5.3 FLP-Aggregates

We will now show that our semantics of aggregates is an extension of the semantics proposed by Faber, Leone and Pfeifer [2004]. An aggregate of the form (2) is a (ground) *FLP-aggregate* if $F_1, \ldots, F_n$ are conjunctions of atoms. A (ground) *FLP-program* is a set of formulas

$$A_1 \wedge \cdots \wedge A_m \supset a_1 \vee \cdots \vee a_n \tag{16}$$

$(n, m \geq 0)$, where $a_1, \ldots, a_n$ are atoms and $A_1, \ldots, A_m$ are FLP-aggregates.[7]

Theorem 3. *The answer sets for a FLP-program under our semantics are identical to its answer sets in the sense of [Faber* et al.*, 2004].*

To prove this theorem we need to observe, first of all, that the definition of satisfaction of FLP-aggregates and FLP-programs in [Faber *et al.*, 2004] is equivalent to ours. The definition of a reduct is different, however. According to [Faber *et al.*, 2004], the *reduct* of a program Π with FLP-aggregates relative to X (we denote such a reduct by $\Pi^{\underline{X}}$) consists of the rules (16) of Π such that $X \models A_1 \wedge \cdots \wedge A_m$. The definition of an answer set is again similar to ours: a set X of atoms is an *answer set* for a FLP-program Π if X is a minimal set satisfying $\Pi^{\underline{X}}$.

Lemma 5. *For any nested expression F without negations and any two sets X and Y of atoms such that $Y \subseteq X$, $Y \models F^X$ iff $Y \models F$.*

Lemma 6. *For any FLP-aggregate $op\langle S \rangle \prec N$ and any set X of atoms, if $X \models op\langle S \rangle \prec N$ then*

$$Y \models (op\langle S \rangle \prec N)^X \text{ iff } Y \models op\langle S \rangle \prec N.$$

Lemma 5 can be proven by structural induction. Lemma 6 follows from Lemma 5 and Proposition 7(b).

Proof of Theorem 3. It is easy to see that if $X \not\models \Pi$ then $X \not\models \Pi^X$ and $X \not\models \Pi^{\underline{X}}$, so that X is not an answer set under either semantics. Now assume that $X \models \Pi$. We will show that the two reducts are satisfied by the same subsets of X. It is sufficient to consider the case in which Π contains only one rule (16). If $X \not\models A_1 \wedge \cdots \wedge A_m$ then $\Pi^{\underline{X}} = \emptyset$, and Π^X is the tautology

$$\bot \supset (a_1 \vee \cdots \vee a_n)^X.$$

Otherwise, $\Pi^{\underline{X}}$ is rule (16), and Π^X is

$$A_1^X \wedge \cdots \wedge A_m^X \supset (a_1 \vee \cdots \vee a_n)^X.$$

These two reducts are satisfied by the same subsets of X by Lemmas 5 and 6. □

[7] The syntax of [Faber *et al.*, 2004] is more general in several ways. An expression of the form $\neg(op\langle S \rangle \prec N)$ in such syntax has the same meaning as $op\langle S \rangle \not\prec N$. Also, that paper allows literals as conjunctive terms in the antecedent of the implication (16). However, semantically, an atom a is not different from $sum\langle\{a = 1\}\rangle \geq 1$, and $\neg a$ is not different from $sum\langle\{a = 1\}\rangle \leq 0$.

6 Conclusion

We extended the definition of an answer set to arbitrary propositional theories. This definition of an answer set is equivalent to the definition of a model in equilibrium logic, so that it shares important properties of equilibrium logic such as the characterization of strong equivalence in terms of the logic of here-and-there. The new definition of reduct is different from the traditional definition [Lifschitz *et al.*, 1999] in the case of programs with nested expressions, but it is in some ways similar to it.

Even though propositional theories have a richer syntax, it turns out that any propositional theory can be expressed as a program with nested expression with the same answer sets [Cabalar and Ferraris, 2005]. In view of this fact, the possibility of defining answer sets for arbitrary propositional theories is not so surprising.

Propositional formulas cover both disjunctive rules with FLP-aggregates and choice rules. In the case of weight constraints, if negative weights are allowed then our semantics is not equivalent to the one of [Simons *et al.*, 2002], but seems to have better properties. We have seen that this difference has consequences from the point of view of computational complexity.

It is possible, by Proposition 7, to view an aggregate $op\langle S\rangle \prec N$ as a primitive construct rather than an abbreviation for an exponentially larger formula. This is what is already happening in the answer set solver DLV [8], which partially supports programs with FLP-aggregates. On the other hand, viewing aggregates as formulas allows us to reason about strong equivalence in terms of the logic of here-and-there.

Acknowledgments

I am grateful to Pedro Cabalar, Selim Erdoğan, Joohyung Lee, David Pearce, Wanwan Ren and Hudson Turner for comments on a previous version of this paper. Special thanks go to Vladimir Lifschitz for many comments and discussions on the topic, and his careful reading of this paper. This research was partially supported by the National Science Foundation under Grant IIS-0412907.

References

[Cabalar and Ferraris, 2005] Pedro Cabalar and Paolo Ferraris. Propositional theories are equivalent to logic programs. In preparation, 2005.

[Calimeri *et al.*, 2005] Francesco Calimeri, Wolfgang Faber, Nicola Leone, and Simona Perri. Declarative and computational properties of logic programs with aggregates. In *Proc. IJCAI'05*, 2005. To appear.

[Eiter and Gottlob, 1993] Thomas Eiter and Georg Gottlob. Complexity results for disjunctive logic programming and application to nonmonotonic logics. In Dale Miller, editor, *Proceedings of International Logic Programming Symposium (ILPS)*, pages 266–278, 1993.

[8] `http://www.dbai.tuwien.ac.at/proj/dlv/` .

[Erdoğan and Lifschitz, 2004] Selim T. Erdoğan and Vladimir Lifschitz. Definitions in answer set programming. In Vladimir Lifschitz and Ilkka Niemelä, editors, *Proceedings of 7th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR-7)*, pages 114–126, 2004.

[Faber *et al.*, 2004] Wolfgang Faber, Nicola Leone, and Gerard Pfeifer. Recursive aggregates in disjunctive logic programs: Semantics and complexity. In *Proc. 9th European Conference on Artificial Intelligence (JELIA'04)*, 2004. Revised version: `http://www.wfaber.com/research/papers/jelia2004.pdf`.

[Ferraris and Lifschitz, 2005] Paolo Ferraris and Vladimir Lifschitz. Weight constraints as nested expressions. *Theory and Practice of Logic Programming*, 5:45–74, 2005.

[Gelfond and Lifschitz, 1988] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In Robert Kowalski and Kenneth Bowen, editors, *Proceedings of International Logic Programming Conference and Symposium*, pages 1070–1080, 1988.

[Gelfond and Lifschitz, 1991] Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991.

[Lifschitz and Turner, 1994] Vladimir Lifschitz and Hudson Turner. Splitting a logic program. In Pascal Van Hentenryck, editor, *Proceedings of International Conference on Logic Programming (ICLP)*, pages 23–37, 1994.

[Lifschitz *et al.*, 1999] Vladimir Lifschitz, Lappoon R. Tang, and Hudson Turner. Nested expressions in logic programs. *Annals of Mathematics and Artificial Intelligence*, 25:369–389, 1999.

[Lifschitz *et al.*, 2001] Vladimir Lifschitz, David Pearce, and Agustin Valverde. Strongly equivalent logic programs. *ACM Transactions on Computational Logic*, 2:526–541, 2001.

[Lifschitz, 1996] Vladimir Lifschitz. Foundations of logic programming. In Gerhard Brewka, editor, *Principles of Knowledge Representation*, pages 69–128. CSLI Publications, 1996.

[Osorio *et al.*, 2004] Mauricio Osorio, Juan Antonio Navarro, and José Arrazola. Safe beliefs for propositional theories. Accepted to appear at Annals of Pure and Applied Logic, 2004.

[Pearce, 1997] David Pearce. A new logical characterization of stable models and answer sets. In Jürgen Dix, Luis Pereira, and Teodor Przymusinski, editors, *Non-Monotonic Extensions of Logic Programming (Lecture Notes in Artificial Intelligence 1216)*, pages 57–70. Springer-Verlag, 1997.

[Pearce, 1999] David Pearce. From here to there: Stable negation in logic programming. In D. Gabbay and H. Wansing, editors, *What Is Negation?* Kluwer, 1999.

[Simons *et al.*, 2002] Patrik Simons, Ilkka Niemelä, and Timo Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138:181–234, 2002.

[Turner, 2003] Hudson Turner. Strong equivalence made easy: nested expressions and weight constraints. *Theory and Practice of Logic Programming*, 3(4,5):609–622, 2003.

An ID-Logic Formalization of the Composition of Autonomous Databases

Bert Van Nuffelen[1], Ofer Arieli[2], Alvaro Cortés-Calabuig[1], and Maurice Bruynooghe[1]

[1] Department of Computer Science, Katholieke Universiteit Leuven, Belgium
{bertv,alvaro,maurice}@cs.kuleuven.ac.be
[2] Department of Computer Science, The Academic College of Tel-Aviv, Israel
oarieli@mta.ac.il

Abstract. We introduce a declarative approach for a coherent composition of autonomous databases. For this we use ID-logic, a formalism that extends classical logic with inductive definitions. We consider ID-logic theories that express, at the same time, the two basic challenges in database composition problems: relating different schemas of the local databases to one global schema (*schema integration*) and amalgamating the distributed and possibly contradictory data to one consistent database (*data integration*). We show that our framework supports different methods for schema integration (as well as their combinations) and that it provides a straightforward way of dealing with inconsistent data. Moreover, this framework facilitates the implementation of database repair and consistent query answering by means of a variety of reasoning systems.

1 Introduction and Motivation

Composition of information that arrives from different data-sources is a major challenge of information systems and its importance has been recognized by many researches. The works on this subject may be divided to two types according to their objectives:

- Systems for *schema integration*, in which the main goal is to provide a uniform vocabulary to which the various vocabularies of the data-sources can be mapped (see, for instance, [6,10,21,26,30]).
- Systems for *data integration*, in which the major concern is to resolve contradictions that may occur when the distributed data is amalgamated (see, e.g., [1,2,3,4,5,8]).

In this work we consider a framework that handles both these tasks *at the same time*. To illustrate (and motivate) this, consider the following situation:

Example 1. Given two data sources. One source stores information about all the students that were enrolled for the first time during 2004, and the other source contains information about all the students whose first year of enrollment is 2005. The encoding of such databases might be the following:

$$\mathcal{DB}_1 = (\{st04(\cdot)\}, \{st04(john)\}),$$
$$\mathcal{DB}_2 = (\{st05(\cdot)\}, \{st05(mary), st05(john)\}).$$

C. Baral et al. (Eds.): LPNMR 2005, LNAI 3662, pp. 132–144, 2005.
© Springer-Verlag Berlin Heidelberg 2005

Here, $\mathcal{DB}_1$ encodes the fact that John is a student first enrolled in 2004, and $\mathcal{DB}_2$ encodes the facts that John and Mary are students first enrolled in 2005.

Assume further that there is a set of (global) integrity constraints, stating that the first year of enrollment is unique for every student, and that someone must have been enrolled already in 2003. That is,

$$\mathcal{IC} = \left\{ \begin{array}{l} \forall XYZ\big(enr(X,Y) \wedge enr(X,Z) \rightarrow Y = Z\big) \\ \exists X enr(X, 2003) \end{array} \right\}.$$

A proper integration system is expected to take the following actions in this case:

a) remove the information that John was first enrolled in 2004 or the information that he was first enrolled in 2005 (*but not both*!)
b) insert a fact that some student *other than John or Mary* was first enrolled in 2003.

Note that in order to accomplish this task, the mediator system should be able to cope with the following challenges:

1. relate the different terminologies of the local databases and that of the global integrity constraints,
2. identify inconsistencies (i.e., violations of integrity constraints) and resolve them by making some (minimal amount of) changes in the unified database,
3. search for solutions that may lie outside the active domain of the databases (in order, e.g., to satisfy the second constraint above).

We call the whole process described above *database composition*. In this paper we present a composition system that generalizes the schema integration process introduced in [30] and at the same time enhances the abductive system for data integration introduced in [3], by using *the same logical formalism*. The outcome is a uniform solution for database composition, which to the best of our knowledge is more comprehensive (with respect to the expressive power of the underlying language, the amount of integration methods that are supported, and the mediation capabilities mentioned in Example 1) than any other approach implemented by similar systems.

2 Preliminaries

2.1 ID-logic

ID-logic [12,13] is a knowledge representation formalism extending classical first-order logic with non-monotone inductive definitions. It is motivated by the realization that (inductive) definitions are a distinctive form of human knowledge and are often encountered in mathematical practice. At the same time, inductive definitions cannot easily be expressed in classical logic (for instance, the transitive closure of a graph is one of the simplest concepts typically defined by induction but it is well-known that this concept cannot be defined in first-order logic).

The language of ID-logic uses the well-founded semantics [28] to extend classical logic with a new 'inductive definition' primitive, and as such it allows even non-monotone inductive definitions to be correctly formalized in an intuitive way. It has also been shown that ID-logic is able to capture the basic ideas behind different concepts and approaches in common-sense reasoning, such as the semantical foundations of situation calculus [15] and description logic [27]. As our goal here is to define composed databases in terms of the distributed ones, together with a description of the merging process, ID-logic is a natural candidate for being the underlying formalism behind such an axiomatization. Below we give the formal definition of this logic.

Definition 1. An ID-logic *theory* $\mathcal{T}$, based on a first-order language $\mathcal{L}$, is a pair ($\mathcal{D}$,$\mathcal{F}$), where $\mathcal{D}$ is a set of definitions D_i $(i = 1, \ldots, n)$ and $\mathcal{F}$ is a set of first-order formulas. A *definition* D is a set of rules of the form $p(\overline{t}) \leftarrow B$, where $p(\overline{t})$ is an atom and B is a first-order formula.

Example 2. The transitive closure of a graph can be defined by the following ID-logic definition: $TransCl(x, y) \leftarrow Edge(x, y) \vee \exists z(TransCl(x, z) \wedge TransCl(z, y))$.

In what follows we refer to $\mathcal{D}$ and $\mathcal{F}$ as the definitions and the constraints (respectively) of $\mathcal{T}$. The predicates occurring in the heads of the rules in the definitions are the *defined* predicates of $\mathcal{D}$. All the other predicates belong to $\mathsf{Open}(\mathcal{D})$, the set of the *open* (*abducible*) predicates of $\mathcal{D}$.

Definition 2. A structure M is a *model* of a definition D iff there exists an interpretation I of $\mathsf{Open}(D)$ such that M is the two-valued well-founded model [28] of D that extends I. A structure M is a model of $\mathcal{D}$ iff M is a model of each $D \in \mathcal{D}$.

Definition 3. A structure M is a *model* of an ID-logic theory $\mathcal{T}$=($\mathcal{D}$,$\mathcal{F}$) iff M is a model of $\mathcal{D}$ and satisfies all formulas of $\mathcal{F}$. The collection of all models of $\mathcal{T}$ is denoted by $mod(\mathcal{T})$.

We say that a formula ψ is *satisfied* by an ID-logic theory $\mathcal{T}$ if there is a model of $\mathcal{T}$ that satisfies ψ. A formula ψ *follows* from $\mathcal{T}$ if every model of $\mathcal{T}$ satisfies ψ.

The following notation will be useful in what follows:

Definition 4. For two ID-logic theories $\mathcal{T}_1, \mathcal{T}_2$ over the same language $\mathcal{L}$, the composed theory $\mathcal{T}_1 \circ \mathcal{T}_2$ is an ID-logic theory $\mathcal{T}$ over $\mathcal{L}$, obtained by the pairwise union of both theories: $\mathcal{T} = \mathcal{T}_1 \circ \mathcal{T}_2 = (\mathcal{D}_1, \mathcal{F}_1) \circ (\mathcal{D}_2, \mathcal{F}_2) = (\mathcal{D}_1 \cup \mathcal{D}_2, \mathcal{F}_1 \cup \mathcal{F}_2)$.

Proposition 1. $mod(\mathcal{T}_1 \circ \mathcal{T}_2) = mod(\mathcal{T}_1) \cap mod(\mathcal{T}_2)$.

2.2 Database Composition

In this section we formally define the problem under consideration and its solutions.

Definition 5. A *database* is a pair $\mathcal{DB} = (\mathcal{L}, \mathcal{D})$, where the *database language* $\mathcal{L}$ is a first-order language based on a vocabulary consisting of the predicate symbols in a fixed database schema S and a finite set Dom of constants representing the elements of the domain of discourse. The *database instance* $\mathcal{D}$ is a finite set of ground atoms in the language $\mathcal{L}$.

The semantics of a database instance is given by the conjunction of the atoms in $\mathcal{D}$ augmented with the *Unique Name Assumption* ($\mathsf{UNA}(\mathsf{Dom})$), i.e., different constants represent different objects, and the *Closed World Assumption* ($\mathsf{CWA}(\mathcal{D})$) that assures that each atom which is not explicitly mentioned in $\mathcal{D}$ is false. Often, the *Domain Closure Assumption* ($\mathsf{DCA}(\mathsf{Dom})$) is also imposed, meaning that all elements of the domain of discourse are named by constants in Dom.[1] The meaning of a database instance under these assumptions is formalized in a model theoretical way by the *least Herbrand model* semantics.

Definition 6. A *composition problem* is a pair $(\mathfrak{D}, \mathfrak{C})$, where the *set of resources* $\mathfrak{D}$ is a non-empty set of (local) databases $\mathcal{DB}_i = (\mathcal{L}_i, \mathcal{D}_i)$, $i = 1, \ldots, n$, and the (global) *integrity constraints* $\mathfrak{C} = (\mathcal{L}_\mathcal{G}, \mathcal{IC})$ consists of a (possibly empty) set $\mathcal{IC}$ of first-order formulae in a first-order (global) language $\mathcal{L}_\mathcal{G}$.

Given a composition problem $(\mathfrak{D}, \mathfrak{C})$, our goal is therefore to construct a composed (global) database $\mathcal{DB}_\mathcal{G} = (\mathcal{L}_\mathcal{G}, \mathcal{D}_\mathcal{G})$ such that its database instance $\mathcal{D}_\mathcal{G}$ contains the translation to $\mathcal{L}_\mathcal{G}$ of data-facts that appear in database instances of elements in $\mathfrak{D}$, provided that these data facts do not violate the integrity constraints in $\mathcal{IC}$. Clearly, this database instance should 'gather' from the local databases as much information as consistently possible, that is, it should be 'as close as possible' to $\bigcup_{i=1,\ldots,n} \mathcal{D}_i$, without violating $\mathcal{IC}$.

Example 3. Consider again Example 1. The composition problem in this case consists of two databases, $\mathcal{DB}_1$ and $\mathcal{DB}_2$, that should be composed under the integrity constraints in $\mathcal{IC}$. When $\mathcal{L}_\mathcal{G} = \{enr(X, Y)\}$ is the language of the composed database (meaning that a student X was first enrolled in a year Y), the two best composed database instances are the following:

$$\mathcal{D}^1_\mathcal{G} = \{\, enr(mary, 2005), enr(john, 2005), enr(u, 2003) \,\}$$
$$\mathcal{D}^2_\mathcal{G} = \{\, enr(mary, 2005), enr(john, 2004), enr(u, 2003) \,\}$$

where u is a (Skolem) constant, different from $mary$ and $john$.

When Dom consists only of $mary$ and $john$, $\mathsf{DCA}(\mathsf{Dom})$ excludes the two solutions above, and the composed database instances are the following (see also Example 5):

$$\mathcal{D}^3_\mathcal{G} = \{\, enr(mary, 2005), enr(john, 2003) \,\}$$
$$\mathcal{D}^4_\mathcal{G} = \{\, enr(john, 2005), enr(mary, 2003) \,\}$$
$$\mathcal{D}^5_\mathcal{G} = \{\, enr(john, 2004), enr(mary, 2003) \,\}$$

Note that other compositions of $\mathcal{DB}_1$ and $\mathcal{DB}_2$ are less intuitive in this case. For instance, $\mathcal{D}^* = \{enr(john, 2003)\}$ requires more revisions in the original assumptions than $\mathcal{D}^3_\mathcal{G}$, and so $\mathcal{D}^3_\mathcal{G}$ is 'closer' to $\mathcal{D}_1 \cup \mathcal{D}_2$ than $\mathcal{D}^*$.

In what follows we shall describe how to define ID-logic theories that *represent* the composition problems under consideration (Section 3) and how to *reason* with these theories in order to *compute* composed databases for the given composition problems (Section 4).

[1] This assumption is sometimes lifted when constants outside Dom should be introduced; see e.g. Examples 3 and 5 below.

3 The Composition Theory

In what follows we assume that all the databases share the same domain Dom (this will simplify the presentation, as the implicit assumptions $\mathsf{UNA}(\mathsf{Dom})$ and $\mathsf{DCA}(\mathsf{Dom})$ can be imposed globally) and that all the languages are mutually distinct (to assure this, one can annotate the predicate names with the source identities).

Definition 7. A *mediator system* for a composition problem $(\mathfrak{D}, \mathfrak{C})$ is a quadruple $\mathcal{M} = \langle \mathcal{L}, \mathsf{CP}, \mathsf{SI}, \mathsf{DI} \rangle$, where:

- $\mathcal{L}$ is the union of the source languages $\mathcal{L}_i$, the global language $\mathcal{L}_{\mathcal{G}}$, and the auxiliary predicates defined below.
- $\mathsf{CP} = \{\mathsf{S}_1, \ldots, \mathsf{S}_n, \mathsf{IC}\}$, *the composition problem description*, is a set of ID-logic theories S_i encoding the source databases (i.e., the database instances $\mathcal{D}_i$ of the databases in $\mathfrak{D}$), and an ID-logic theory IC encoding the global integrity constraints $\mathcal{IC}$ in $\mathfrak{C}$ (see Section 3.1).
- $\mathsf{SI} = \{\mathsf{M}_1, \ldots, \mathsf{M}_n, \mathsf{K}\}$, *the schema integration specification*, is a set of ID-logic theories M_i encoding the relations between the source languages $\mathcal{L}_i$ and the global language $\mathcal{L}_{\mathcal{G}}$, and an ID-logic theory K encoding additional information about the relations among the schemas (see Section 3.2).
- $\mathsf{DI} = \{\mathsf{Comp}, \mathsf{Trans}\}$, *the data integration specification*, is a set of ID-logic theories that specify how to make the combined database consistent with the set $\mathcal{IC}$ of integrity constraints in $\mathfrak{C}$ (see Section 3.3).

A *composition theory* for a mediator system $\mathcal{M}$ is an ID-logic theory $\mathcal{T}_{\mathcal{M}} = \mathcal{T}_{\mathsf{SI}} \circ \mathcal{T}_{\mathsf{DI}}$ in $\mathcal{L}$, where $\mathcal{T}_{\mathsf{SI}} = \mathsf{S}_1 \circ \cdots \circ \mathsf{S}_n \circ \mathsf{M}_1 \circ \cdots \circ \mathsf{M}_n \circ \mathsf{K}$ and $\mathcal{T}_{\mathsf{DI}} = \mathsf{IC} \circ \mathsf{Comp} \circ \mathsf{Trans}$.

In the rest of this section we consider in further details the components of a mediator.

3.1 Representation of the Composition Problem

Let $(\{(\mathcal{L}_1, \mathcal{D}_1), \ldots, (\mathcal{L}_n, \mathcal{D}_n)\}, (\mathcal{L}_{\mathcal{G}}, \mathcal{IC}))$ be a composition problem.

- The encoding of a source database instance $\mathcal{D}_i$ $(1 \leq i \leq n)$ in CP is the ID-logic theory $\mathsf{S}_i = (\{\{D_i\}\}, \emptyset)$, where D_i is the enumeration of the facts in $\mathcal{D}_i$.
- The encoding of the global integrity constraints $\mathcal{IC}$ in CP is the ID-logic theory $\mathsf{IC} = (\emptyset, \{IC\})$, where IC is obtained by interpreting the integrity constraints $\mathcal{IC}$ as an enumeration of formulae in $\mathcal{L}_{\mathcal{G}}$, and substituting every occurrence of a predicate $p(\overline{t})$ by $fact(p(\overline{t}))$.[2]

3.2 Representation of Schema Integration

The component SI of a mediator system describes the relationships between the source languages $\mathcal{L}_i$ and the global language $\mathcal{L}_{\mathcal{G}}$. These relationships are expressed in the form of (inductive) definitions, taking into account the ontological relationships between the predicates and the actual knowledge of the source. For a proper description of such relations, one has to take into consideration the following cases:

[2] The need of this substitution will become apparent in Section 3.3.

1. Suppose that the set of rules $\{p(\overline{t}) \leftarrow B_i \mid i = 1 \dots k\}$ only partially defines a predicate p. A complete definition can be obtained by adding a rule $p(\overline{s}) \leftarrow p^*(\overline{s})$, in which the auxiliary open predicate p^* represents all the tuples in p that are not defined by any of the bodies B_i.
2. Sometimes a body of a rule $p(\overline{t}) \leftarrow B$ is too general, i.e., it includes tuples not intended to be in the predicate p. In this case it is possible to add to the body B an auxiliary open predicate p^s that filters the extraneous tuples. The completed rule in this case is $p(\overline{t}) \leftarrow B \wedge p^s(\overline{t})$.

Auxiliary predicates help to relate different languages and to complete partial information. As such, they serve as open (abducible) predicates.

Definition 8. A *language mapping* from a language $\mathcal{L}_1$ to a language $\mathcal{L}_2$ is an ID-logic theory $(\mathsf{R}_{1\to 2}, \mathsf{IC}_{1,2})$, where $\mathsf{R}_{1\to 2}$ defines the predicates of $\mathcal{L}_2$ in terms of the predicates of $\mathcal{L}_1$ and the necessary auxiliary predicates, and $\mathsf{IC}_{1,2}$ formulates generic integrity constraints on the auxiliary predicates.

In terms of the last definition, the elements $\mathsf{M}_1, \dots, \mathsf{M}_n$ of SI are the language mappings between the source languages and the global language. The first component of each one of these mappings M_i may be of one of the following two forms:

- $\mathsf{R}_{i\to\mathcal{G}}$. In this case the predicates of global language $\mathcal{L}_{\mathcal{G}}$ are defined in terms of the predicates of a local language ($\mathcal{L}_i$ in this case). This mapping corresponds to the *Global as View* (GAV) approach in schema integration (see, e.g., [26]).
- $\mathsf{R}_{\mathcal{G}\to i}$. In this case the language $\mathcal{L}_i$ of a local source is defined in terms of the global language $\mathcal{L}_{\mathcal{G}}$. This mapping corresponds to the *Local as View* (LAV) approach in schema integration (see [21]).

Most of the systems that are introduced in the literature support only one type of the schema integration methods described above. In our case it is clear that one may use both kinds of language mappings in the same theory, and so imitate both GAV and LAV by the same mediator system.

Example 4. Consider the source languages $\mathcal{L}_1 = \{st05(\cdot)\}$ and the global language $\mathcal{L}_{\mathcal{G}} = \{enr(\cdot,\cdot)\}$, where $st05(\cdot)$ represents (some) students that are first enrolled in 2005 and $enr(\cdot,\cdot)$ represents all students with their first year of enrollment. According to the LAV approach, a mapping $\mathsf{M}_1 = (\mathsf{R}_{\mathcal{G}\to 1}, \mathsf{IC}_{\mathcal{G},1})$ between these languages could be the following ID-logic theory:

$$\begin{pmatrix} \{\{st05(X) \leftarrow enr(X, 2005) \wedge st05^s(X).\}\}, \\ \{\forall X(st05^s(X) \rightarrow enr(X, 2005))\} \end{pmatrix},$$

where $st05^s(\cdot)$ represents the students known by the source and the integrity constraint expresses that this is a subset of all students enrolled in 2005. Similarly, a GAV mapping $\mathsf{M}_1 = (\mathsf{R}_{1\to\mathcal{G}}, \mathsf{IC}_{1,\mathcal{G}})$ could be, e.g., the following ID-logic theory:

$$\begin{pmatrix} \{\{enr(X,Y) \leftarrow (st05(X) \wedge Y = 2005) \vee enr^*(X,Y).\}\}, \\ \forall XY(enr^*(X,Y) \rightarrow \neg(st05(X) \wedge Y = 2005)) \end{pmatrix},$$

where $enr^*(\cdot,\cdot)$ represents the students not known by the source, and the integrity constraint imposes that $enr^*(\cdot,\cdot)$ does not duplicate the information from the source.

The remaining element in SI, denoted by K, contains some additional information about the predicates and the interrelations. In the above example, for instance, one may know that the source has complete information about all students enrolled in 2005, so now $st05(\cdot)$ should represent *all* the students first enrolled in 2005. This can be expressed by the constraint $\forall X enr(X, 2005) \leftrightarrow st05(X)$. In the LAV approach it implies that the relation $st05^s(\cdot)$ is empty (and could be omitted in the language mapping); in the GAV approach, it implies that $enr^*(\cdot,\cdot)$ cannot have tuples with the year 2005.

3.3 Representation of Data Integration

The data integration specification DI is a specification of how the database instances should be integrated such that no integrity constraint will be violated. It makes sure that if the set $\mathcal{D} = \bigcup \mathcal{D}_i$ of all the local databases (translated to the global schema) preserves all the integrity constraints in $\mathcal{IC}$ (notation: $\mathcal{D} \models \mathcal{IC}$),[3] then the global database instance $\mathcal{D}_{\mathcal{G}}$ will be equal to this set. Otherwise, some (minimal amount of) data-facts should be inserted to- or retracted from this union in order to restore its consistency with respect to $\mathcal{IC}$. In other words, $\bigcup \mathcal{D}_i$ should be 'repaired':

Definition 9. [1] A *repair* of a database instance $\mathcal{D}$ with respect to a set $\mathcal{IC}$ of integrity constraints is a pair $(\mathsf{Insert}, \mathsf{Retract})$, such that: (1) $\mathsf{Insert} \cap \mathcal{D} = \emptyset$, (2) $\mathsf{Retract} \subseteq \mathcal{D}$,[4] and (3) $((\mathcal{D} \cup \mathsf{Insert}) \setminus \mathsf{Retract}) \models \mathcal{IC}$.

Intuitively, Insert is a set of elements that should be inserted to $\mathcal{D}$ and Retract is a set of elements that should be removed from $\mathcal{D}$ in order to assure that $\mathcal{D}$ is consistent with $\mathcal{IC}$. This is represented by the following ID-logic theory (the *composer*):

$$\mathsf{Comp} = \left(\begin{array}{l} \left\{ \left\{ \begin{array}{l} fact(X) \leftarrow db(X) \wedge \neg retract(X). \\ fact(X) \leftarrow insert(X). \end{array} \right\} \right\}, \\ \left\{ \begin{array}{l} \forall X \, \neg(insert(X) \wedge db(X)) \\ \forall X \, db(X) \leftarrow retract(X) \end{array} \right\} \end{array} \right),$$

where $db(X)$ denotes in the global language that X is a data-fact, and $fact(X)$ denotes that the data-fact X should appear in the global database. Here, $insert$ and $retract$ are open (abducible) predicates that describe repairs. The last two formulas of Comp assure that conditions (1) and (2) in Definition 9 will hold.[5]

As there are usually many ways to repair a given database, it is often convenient to make preferences among the possible repairs, and consider only the most preferred ones. Below are two common preference criteria for preferring a repair $(\mathsf{Insert}, \mathsf{Retract})$ over a repair $(\mathsf{Insert}', \mathsf{Retract}')$:

Definition 10. Let $(\mathsf{Insert}, \mathsf{Retract})$ and $(\mathsf{Insert}', \mathsf{Retract}')$ be two repairs of a database.

- *set inclusion preference criterion*:
  $(\mathsf{Insert}, \mathsf{Retract}) \leq_i (\mathsf{Insert}', \mathsf{Retract}')$, if $\mathsf{Insert} \subseteq \mathsf{Insert}'$ and $\mathsf{Retract} \subseteq \mathsf{Retract}'$

[3] That is, every formula in $\mathcal{IC}$ is satisfied in the least Herbrand model of $\mathcal{D}$.
[4] Note that by conditions (1) and (2), $\mathsf{Insert} \cap \mathsf{Retract} = \emptyset$.
[5] The third condition of Definition 9 is imposed by the theory IC as defined in Section 3.1.

– *minimal cardinality preference criterion*:
$(\mathsf{Insert}, \mathsf{Retract}) \leq_c (\mathsf{Insert}', \mathsf{Retract}')$, if $|\mathsf{Insert}|+|\mathsf{Retract}| \leq |\mathsf{Insert}'|+|\mathsf{Retract}'|$

If $\mathcal{D} \models \mathcal{IC}$, then $(\emptyset, \emptyset)$ is the *only* $\leq_i$- and $\leq_c$-preferred repair of $\mathcal{D}$, as expected.

The second component of DI is the *translator*, Trans. It represents all the translated data-facts in terms of one (global) language:

$$\mathsf{Trans} = (\{\{db(p_1(\bar{t})) \leftarrow p_1(\bar{t}), \ldots, db(p_{\mathcal{G}}(\bar{t})) \leftarrow p_{\mathcal{G}}(\bar{t})\}\}, \emptyset)$$

where $p_1, \ldots, p_{\mathcal{G}}$ are the predicates of the global language $\mathcal{L}_{\mathcal{G}}$.

The translator reifies the database predicates, i.e., it converts the database facts into terms of the predicate db.[6] For reducing notational complexity, we use the same symbols for the predicates and their reifications (e.g., in Trans above, the p_i appearing on the left-hand side of the implications are the reified symbols of the predicates p_i on the right-hand side of the same implications). Note that the predicate fact of the composer represents which one on these new 'fact'-terms appears in the composed database.

3.4 Back to the Canonical Example

A composition theory for Example 1 may be the following:

- The schema integration theory $\mathcal{T}_{\mathsf{SI}}$ is a composition of S_1, S_2,M_1, and M_2 (in this case K is assumed to be empty):

$$\left(\left\{ \begin{array}{l} \{ st04(john). \} \\ \left\{ \begin{array}{l} st05(mary). \\ st05(john). \end{array} \right\} \\ \left\{ \begin{array}{l} enr(X,Y) \leftarrow st04(X) \wedge Y = 2004. \\ enr(X,Y) \leftarrow enr_1^*(X,Y). \end{array} \right\} \\ \left\{ \begin{array}{l} enr(X,Y) \leftarrow st05(X) \wedge Y = 2005. \\ enr(X,Y) \leftarrow enr_2^*(X,Y). \end{array} \right\} \end{array} \right\}, \left\{ \begin{array}{l} \forall XY(enr_1^*(X,Y) \rightarrow \neg(st04(X) \wedge Y = 2004)). \\ \forall XY(enr_2^*(X,Y) \rightarrow \neg(st05(X) \wedge Y = 2005)). \end{array} \right\} \right)$$

The definitions of $\mathcal{T}_{\mathsf{SI}}$ are from the theories S_1, S_2, M_1, and M_2; the constraints are from M_1 and M_2.

- The data integration theory $\mathcal{T}_{\mathsf{DI}}$ is the following composition of IC, Comp and Trans:

$$\left(\left\{ \begin{array}{l} \left\{ \begin{array}{l} fact(X) \leftarrow db(X) \wedge \neg retract(X). \\ fact(X) \leftarrow insert(X). \end{array} \right\} \\ \{ db(enr(X,Y)) \leftarrow enr(X,Y). \} \end{array} \right\}, \left\{ \begin{array}{l} \forall X \neg(insert(X) \wedge db(X)). \\ \forall X db(X) \leftarrow retract(X). \\ \forall XYZ\ (fact(enr(X,Y)) \wedge fact(enr(X,Z)) \rightarrow Y = Z). \\ \exists X fact(enr(X, 2003)). \end{array} \right\} \right)$$

The definitions of $\mathcal{T}_{\mathsf{DI}}$ are from Comp (the first two) and Trans (the third one); the constraints of $\mathcal{T}_{\mathsf{DI}}$ are from Comp (the first two) and IC (the last two).

[6] See [7] for a description of reifications in the context of knowledge representation.

4 Reasoning with Composed Databases

Query answering is probably the main task of a mediator system. In order to compute answers from a composition theory in our context, the underlying ID-logic theory should be converted to an equivalent theory in answer set programming (ASP) or abductive logic programming (ALP), which are the two available methods of reasoning with ID-logic theories. By this, corresponding off-the-shelf solvers (such as the ASP systems dlv [17] and sModels [25], or the ALP solver $\mathcal{A}$system [3,20,29]) can be utilized for the query answering. Below we consider both options.

Abductive Logic Programming

An ID-logic theory can be converted to an equivalent abductive normal logic program (see [29] for a detailed description of this process), and then processed by solvers for reasoning with abductive theories. We have implemented our approach for database composition by such a solver, called $\mathcal{A}$system [3,20,29].[7] The $\mathcal{A}$system computes interpretation of the abducible predicates of a given ID-logic theory[8] by executing the abductive refutation procedure SLDNFA (an extension of **SLD**-resolution for programs with **N**egation as **F**ailure operators and **A**bducible predicates; see [14]). It therefore constructs an explanation formula $\mathcal{E}$, in terms of the open predicates of $\mathcal{T}$, that entails a query $\mathcal{Q}$. Formally:

Definition 11. An *abductive solution* for an ID-logic theory $\mathcal{T}$ and a query $\mathcal{Q}$ is a pair $(\Delta, \mathcal{E})$, where Δ is a set of abducible atoms and $\mathcal{E}$ is the conjunction of the elements in Δ, such that $\mathcal{T} \models \exists \overline{x}\, \mathcal{E}(\overline{x})$ and $\mathcal{T} \models \forall \overline{x}\, (\mathcal{E} \rightarrow \mathcal{Q})(\overline{x})$.

In our case, the computed explanation formula $\mathcal{E}$ describes a class of models of $\mathcal{T}$. When $\mathcal{E}$ is *true*, the query is satisfiable with respect to all the models. When the $\mathcal{A}$system is unable to find an abductive solution for $\mathcal{Q}$, then $\mathcal{T} \models \forall(\neg \mathcal{Q})$.

SLDNFA is a sound proof procedure for abductive normal logic programs under the (three-valued) completion semantics. Under certain conditions it is also complete (see [14]) and always terminates (see [31]).[9] These properties are inherited by the $\mathcal{A}$system, which is also equipped with a component that discards non-optimal solutions, called the *optimizer*. Given a preference criterion on the solution space, the optimizer computes only the most-preferred (abductive) solutions by pruning 'on the fly' those branches of the search tree that lead to solutions that are worse than others that have already been computed. This is actually a branch and bound 'filter' that speeds-up execution and makes sure that only the desired solutions will be obtained. If the preference criterion is a pre-order (as those of Definition 10), the optimizer is *complete*, that is, it can compute all the optimal solutions (as illustrated in Example 5 below). We refer to [3,29] for a detailed description of the abductive inference process implemented by the $\mathcal{A}$system.

---

[7] See also `http://www.cs.kuleuven.ac.be/~dtai/kt/systems-E.shtml`.

[8] These interpretations uniquely determine the models of the theory; see Definitions 2 and 3.

[9] This is the case, for instance, when the underlying logic programs are hierarchical, or abductive non-recursive

Example 5. Consider again the composition theory $\mathcal{T}_{\mathcal{M}} = \mathcal{T}_{\mathsf{SI}} \circ \mathcal{T}_{\mathsf{DI}}$ given in Section 3.4 for the running example. By $\mathcal{T}_{\mathsf{SI}}$ we derive the atoms $enr(john, 2004)$, $enr(john, 2005)$, $enr(mary, 2005)$, which is the translation of the local data in terms of the global language. Now, both integrity constraints in $\mathcal{IC}$ are violated, so the data should be repaired. Indeed, by $\mathcal{T}_{\mathsf{DI}}$ and $\leq_c$-optimizer, the following repairs are obtained:

- $retract(enr(john, 2004))$, $insert(enr(u, 2003))$ for $u \notin \{john, mary\}$,
- $retract(enr(john, 2005))$, $insert(enr(u, 2003))$ for $u \notin \{john, mary\}$.

With an $\leq_i$-optimizer, three more solutions are obtained:

- $retract(enr(john, 2004)), retract(enr(john, 2005)), insert(enr(john, 2003))$
- $retract(enr(mary, 2005)), retract(enr(john, 2004)), insert(enr(mary, 2003))$
- $retract(enr(mary, 2005)), retract(enr(john, 2005)), insert(enr(mary, 2003))$

The global database instances in this case are, respectively,

- $\{enr(u, 2003)), enr(john, 2005)), enr(mary, 2005))\}$,
- $\{enr(u, 2003)), enr(john, 2004)), enr(mary, 2005))\}$.
- $\{enr(john, 2003)), enr(mary, 2005))\}$.
- $\{enr(mary, 2003)), enr(john, 2005))\}$.
- $\{enr(mary, 2003)), enr(john, 2004))\}$.

The first two solutions are obtained since the $\mathcal{A}$system does not impose the domain closure assumption $\mathsf{DCA}(\mathsf{Dom})$. This allows to compute solutions outside the least Herbrand model of the problem, and so to suggest explanations for database inconsistency, which could not be captured otherwise.

Answer Set Programming

An ID-logic theory in which each variable occurring in a formula is delimited by a range (domain) relation is called a strongly range-restricted theory. In [23] it is shown that strongly range-restricted ID-logic theories can be transformed to equivalent logic programs under the stable model semantics. This implies that ASP solvers may also be incorporated for reasoning with ID-logic-based mediator systems.

ALP and ASP have a lot in common, and they are often viewed as different variations of the same paradigm. In particular, both approaches compute (minimal) models of the theory. Still, in opposed to the ALP approach, which is a local inference procedure that selects only the information which is relevant for the query, ASP is a global reasoning tool for processing ground theories. ASP requires finite domains and imposes the domain closure axiom. As a consequence, this method is conceptually less suitable for reasoning about tasks which need to go outside the Herbrand space, and it is inherently less scalable (in terms of the size of the databases) than ALP.[10] Note, however, that grounding to finite theories ensures the termination of the ASP computations.

[10] In Example 5, for instance, ASP solvers will not produce the two repairs that contain the Skolem constant u.

A simple work around that allows to lift the domain closure assumption posed by the ASP approach is to iteratively add new Skolem constants to the Herbrand domain and check for solutions in the new domains. However, a problem with this naive generate-and-test approach is that one needs a criterion to know whether all the solutions have been found. In case of a preference condition, the number of the Skolems constants used in a solution allows to derive a lower bound on its cost, which could likely be the basis for a terminating condition. Note that in certain cases (e.g., when no insertions are allowed to restore consistency), all the solutions are already inside the Herbrand domain, and so ASP and ALP solvers will terminate.

5 Concluding Remarks

In this paper we have developed a formal declarative foundation for representing and reasoning with independent databases that contain information about a common domain, but may have different schemas and may contradict each other. This problem, known as database composition, is represented by ID-logic theories that mediate among the ontologies of the sources, and resolve contradictions between local information and global constraints.

It is important to note that this paper is mainly concerned with the *representation* aspects of this problem, showing that different ingredients of it can be expressed in a natural and intuitive way by a single logical formalism. In this context, we have elaborated on the following advantages of our approach:

- The underlying logic extends classical logic with inductive definitions, and as such it can be viewed as an expressive form of a description logic.[11] In particular, our approach is more expressive than similar approaches that are based on description logics.
- Unlike some other approaches of data integration, no syntactical restriction is imposed on the integrity constraints (which can be *any* set of first-order formulas).
- The inherent modularity of ID-logic allows to represent different aspects of the same problem (that is, schema and data integration) in different modules. In other formalisms (e.g., ALP, ASP, or description logics) these aspects are mixed in one complex theory.
- The representation methodology is tolerant to the structure of the autonomous databases. For instance, the composition theory described in Section 3 may be easily modified in case that a certain source of information is added or dropped.
- Different types of schema integration are supported (e.g., GAV and LAV), as well as their combinations and corresponding extensions, such as the generalized LAV approach (GLAV) [19] and Both-as-View approach (BAV) [24].

Other benefits of our framework, which are related to *computation* aspects of data integration, are hinted in Section 4. Below we list two of them: [12]

[11] See [27] for more information about the relation between ID-logic and description logics.

[12] The full details are beyond the scope (and the space limitations) of this paper. Still, as noted in Section 4, the properties below are obtained by straightforward generalizations or adaptations to our context of the techniques described in [3] (for ALP) and [23] (for ASP).

– Different types of query answering are supported. I.e., *skeptical query answering* (also called *certain answering*), in which a query is true iff it is entailed by *every* composed database, or *credulous query answering*, in which a query is true iff it is entailed by *some* composed database.
– Different notions of optimal repairs (e.g., set inclusion, minimal cardinality, minimization of the amount of inserted data-facts, and so forth) are dealt with through preferential semantics.

As noted above, the mediator systems considered here may be implemented by a variety of off-the-shelf solvers. Several other implementations have been introduced for the kind of problems we are dealing with here. Among the implementations of schema integration are the abductive GAV-based system of [9] and the LAV-based information manifold system of [21]. Systems for data integration are, e.g., BReLS [22] and the data repair system of [18]. We provide here a uniform framework for *both* kinds of integrations. Recently, some other implementations of schema and data integration have been introduced, e.g., [8] and [10]. These approaches are based on representation platforms that are more restricted than ours, as they implement only particular kinds of schema mapping styles, limit the syntactic structure of the integrity constraints, and impose the domain closure assumption.

A detailed investigation of the properties of particular computational models for our framework is beyond the scope of the current paper. We refer to [11,16] for a discussion on some computational aspects (e.g, complexity and decidability) of the kinds of problems considered here. Other topics for future elaboration include incorporation of temporal information in the databases, handling of conflicts among integrity constraints, and a study of other merging policies (such as merging by majority vote).

References

1. M. Arenas, L. Bertossi, and J. Chomicki. Consistent query answers in inconsistent databases. In *Proc. PODS'99*, pages 68–79, 1999.
2. M. Arenas, L. Bertossi, and J. Chomicki. Answer sets for consistent query answering in inconsistent databases. *Theory and Practice of Logic Programming*, 3(4–5):393–424, 2003.
3. O. Arieli, M. Denecker, B. Van Nuffelen, and M. Bruynooghe. Coherent integration of databases by abductive logic programming. *Artif. Intelligence Research*, 21:245–286, 2004.
4. O. Arieli, M. Denecker, B. Van Nuffelen, and M. Bruynooghe. Database repair by signed formulae. In *Proc. FoIKS'04*, LNCS 2942, pages 14–30, 2004.
5. C. Baral, S. Kraus, and J. Minker. Combining multiple knowledge bases. *IEEE Trans. on Knowledge and Data Enginnering*, 3(2):208–220, 1991.
6. C. Batini, M. Lenzerini, and S. B. Navathe. A comparative analysis of methodologies for database schema integration. *ACM Computing Surveys*, 18(4):323–364, 1986.
7. R. Brachman and H. Levesque. *Knowledge representation and Reasoning*. Morgan Kaufmann, 2004.
8. L. Bravo and L. Bertossi. Logic programming for consistently querying data integration systems. In *Proc. IJCAI'03*, pages 10–15, 2003.
9. S. Bressan, C. H. Goh, K. Fynn, M. Jakobisiak, K. Hussein, H. Kon, T. Lee, S. Madnick, T. Pena, J. Qu, A. Shum, and M. Siegel. The context interchange mediator prototype. In *Proc. PODS'97*, pages 525–527. ACM, 1997.

10. A. Calì, D. Calvanese, G. De Giacomo, and M. Lanzerini. Data integration under integrity constraints. *Information Systems*, 29(2):147–163, 2004.
11. A. Calì, D. Lembo, and R. Rosati. On the decidability and complexity of query answering over inconsistent and incomplete databases. In *Proc. PODS'03*, pages 260–271. ACM, 2003.
12. M. Denecker. Extending classical logic with inductive definitions. In *Proc. CL'2000*, LNCS 1861, pages 703–717. Springer, 2000.
13. M. Denecker, M. Bruynooghe, and V. Marek. Logic programming revisited: logic programs as inductive definitions. *ACM Trans. on Computational Logic*, 2(4):623–654, 2001.
14. M. Denecker and D. De Schreye. SLDNFA an abductive procedure for abductive logic programs. *Journal of Logic Programming*, 34(2):111–167, 1998.
15. M. Denecker and E. Ternovska. Inductive situation calculus. In *Proc. KR'04*, pages 545–553. Morgan Kaufmann Publishers, 2004.
16. T. Eiter, M. Fink, G. Greco, and D. Lembo. Efficient evaluation of logic programs for querying data integration systems. In *Proc. ICLP'03*, LNCS 2916, pages 163–177. Springer, 2003.
17. T. Eiter, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. The KR system dlv: Progress report, comparisons and benchmarks. In *Proc. KR'98*, pages 406–417, 1998.
18. E. Franconi, A. Palma, N. Leone, D. Perri, and F. Scarcello. Census data repair: A challenging application of disjunctive logic programming. In *Proc. LPAR'01*, LNCS 2250, pages 561–578. Springer, 2001.
19. M. Friedman, A. Y. Levy, and T. D. Millstein. Navigational plans for data integration. In *Proc. 16th. AAAI*, pages 67–73, 1999.
20. A. Kakas, B. Van Nuffelen, and M. Denecker. A-system : Problem solving through abduction. In *Proc. IJCAI'01*, pages 591–596, 2001.
21. A. Levy, A. Rajaraman, and Ordille J.J. Querying heterogeneous information sources using source descriptions. In *Proc. VLDB-96*, pages 251–262, 1996.
22. P. Liberatore and M. Schaerf. BReLS: A system for the integration of knowledge bases. In *Proc. KR'2000*, pages 145–152. Morgan Kaufmann Publishers, 2000.
23. M. Mariën, D. Gilis, and M. Denecker. On the relation between ID-logic and answer set programming. In *Proc. JELIA'04*, LNCS 3229, pages 108–120. Springer, 2004.
24. P. McBrien and A. Poulovassilis. Data integration by bi-directional schema transformation rules. In *Proc. 19th ICDE*. IEEE, 2003.
25. P. Simons, I. Niemelä, and T. Soinien. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1–2), 2002.
26. J. Ullman. Information integration using logical views. *Theoretical Computer Science*, 239(2):189–210, 2000.
27. K. Van Belleghem, M. Denecker, and D. De Schreye. A strong correspondence between description logics and open logic programming. In *Proc. ICLP'97*, pages 346–360, 1997.
28. A. Van Gelder, K. A. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.
29. B. Van Nuffelen. *Abductive constraint logic programming*. PhD thesis, Katholieke Universiteit Leuven, 2004.
30. B. Van Nuffelen, A. Cortés-Calabuig, M. Denecker, O. Arieli, and M. Bruynooghe. Data integration using ID-logic. In *Proc. CAiSE'04*, LNCS 3084, pages 67–81. Springer, 2004.
31. S. Verbaeten. Termination analysis for abductive general logic programs. In *Proc. ICLP'99*, pages 365–379. MIT Press, 1999.

On the Local Closed-World Assumption of Data-Sources*

Alvaro Cortés-Calabuig[1], Marc Denecker[1], Ofer Arieli[2], Bert Van Nuffelen[1], and Maurice Bruynooghe[1]

[1] Department of Computer Science, Katholieke Universiteit Leuven, Belgium
{Alvaro, Marc.Denecker, Bert.VanNuffelen, Maurice.Bruynooghe}@cs.kuleuven.ac.be

[2] Department of Computer Science, The Academic College of Tel-Aviv, Israel
oarieli@mta.ac.il

Abstract. The *Closed-World Assumption* (CWA) on a database expresses that an atom not in the database is false. The CWA is only applicable in domains where the database has complete knowledge. In many cases, for example in the context of distributed databases, a data source has only complete knowledge about part of the domain of discourse. In this paper, we introduce an expressive and intuitively appealing method of representing a *local closed-world assumption* (LCWA) of autonomous data-sources. This approach distinguishes between the data that is conveyed by a data-source and the meta-knowledge about the area in which these data is complete. The data is stored in a relational database that can be queried in the standard way, whereas the meta-knowledge about its completeness is expressed by a first order theory that can be processed by an independent reasoning system (for example a *mediator*). We consider different ways of representing our approach, relate it to other methods of representing local closed-word assumptions of data-sources, and show some useful properties of our framework which facilitate its application in real-life systems.

1 Introduction and Motivation

In recent years, information integration has attracted considerable attention from the AI and databases communities. Generally speaking, the idea is, given a set of independent data-sources, to characterize the collective knowledge represented by them in terms of a uniform vocabulary, called the global schema, and then to exploit this information to obtain correct answers from the whole system (see [8] for a detailed description of this problem, and [1,9,14] for some particular solutions for it). An important aspect of this research is to arrive at an exact description of the information endorsed by each and every data-source in the system. Typically, a data-source stores a database consisting of a set of tuples. In standard database settings, the information held by the data-source would be

* This work is supported by FWO-Vlaanderen, European Framework 5 Project WASP, and by GOA/2003/08.

C. Baral et al. (Eds.): LPNMR 2005, LNAI 3662, pp. 145–157, 2005.
© Springer-Verlag Berlin Heidelberg 2005

expressed by the conjunction of atoms together with the *closed-world assumption* [13]. The CWA expresses the *communication agreement* that an atom that does not appear in the database is false. However, it is clear that the CWA can only be applied when the database contains complete knowledge of the domain of discourse. In a context of distributed data-sources this assumption is inherently inappropriate since a consideration of a certain data-source as a single and complete representation of the world either completely discards the other sources of information or causes contradictions among them. For this reason, some existing approaches have applied an *open-world assumption* [3,11], interpreting a data-source just as the conjunction of atoms in the database. However, we find that this does not allow to grasp more refined information that is held in distributed data systems (sometimes called mediator-based systems). We illustrate this in the following example.

Example 1. Consider a distributed traffic tax administration system, in which there is one data-source for each county, maintaining a database of car owners in that county. There is a protocol amongst the different counties so that when a car owner leaves one county $\mathcal{A}$ to live in another county $\mathcal{B}$, then county $\mathcal{A}$ immediately transfers its information to county $\mathcal{B}$, while still preserving a record of the car owner and its current status for a certain period of time, to handle all running tax demands. By the nature of the protocol, we may assume that each data-source has complete knowledge about all car owners in its county, but in general it has more information than that. Part of the tables of a particular county, say Bronx, may look as follows:

Car Owners

Name	Model	CarID
Peter Steward	Mercedes 320	Qn-5452
John Smith	Volvo 230	Bx-5242
Mary Clark	BMW 550	Bx-5462

Location

Name	Residence
Peter Steward	Queens
Mary Clark	Bronx
John Smith	Bronx

By the nature of the distributed system, this data-source has an expertise on car owners of Bronx. This meta-knowledge allows to derive further information that is not explicitly stated in the data-source, e.g. that all people that are recorded in the table Location as residents of Bronx, are actually *all* the car owners from that county. However, as the information about car owners in Queens is not complete in this data-source, one should not rely only on the tables of this source for making further conclusions about that county.

The example above shows that when the information is distributed over several independent data-sources, a different approach is needed in order to properly capture the meaning of a particular data-source. While in distributed information systems, data-sources usually have only partial knowledge about the domain of discourse, still it is often the case that a particular source is an expert about a specific area and has complete knowledge about it. We call this area the *window of expertise* of the data source. It follows that to express the *information*

held by a data source, the explicit data recorded in the data-source has to be complemented by a meta-information that describes this window of expertise. These two kinds of information should be separated as much as possible, so that one may still consider data-sources as relational databases, and process the information about their completeness by an independent reasoning system.

Following these guidelines, we represent the meta-information about the completeness of a data-source by a theory that consists of several *local closed-world assumptions* (LCWA) [4]. A LCWA refines the closed-world assumption by specifying for a certain predicate an area in which the data source contains all true tuples of the predicate. In our approach, the semantics of a LCWA is expressed by a first-order formula of a uniform syntactical form. Specifically, the contribution of this paper is the following:

- A new method for representing local closed-world assumptions is introduced. Unlike other methods for expressing such assumptions, conceived e.g. in [2,4], which are tailored for intelligent agents, our notion of LCWA is specifically devised for describing complete knowledge in relational data-sources that are part of mediator systems. This allows, in particular, to formally define the *meaning* of each and every data source in such systems.
- The representation of the local closed-world assumption considered here allows to distinguish between the explicit data of the source and the external (implicit) information about its completeness. This separation allows to query a data-source in the standard way, whereas the knowledge about its completeness can be independently processed by the mediator system.
- We present two equivalent representations of the *meaning* of data-sources. One representation is given in terms of first-order theories and the other one is based on circumscription [10] (which is the common approach for expressing LCWA in related works; see, e.g., [2]). This equivalence allows us to show how our proposal captures the intuition behind traditional approaches for LCWA, expressed in terms of higher-order languages, and how they can be reduced to first-order theories in case that certain conditions are met.

The organization of the rest of paper is the following. In Section 2 we introduce the local closed-world assumption and use it for defining the meaning of a data-source. In Section 3 we consider an alternative approach, defined in terms of second-order, pseudo-circumscriptive formulae, and show the equivalence between the two approaches. Then, in Section 4 we give some further comments and generalizations to the local closed-world assumption, and in Section 5 we discuss some other approaches to this assumption. Section 6 concludes the paper.

2 The Local Closed-World Assumption (LCWA)

Definition 1. A *data-source* S is a pair $\langle \Sigma, D \rangle$, where Σ is a vocabulary consisting of predicate symbols in a fixed relational schema $\mathcal{R}(\Sigma)$ and a finite set

$\mathcal{C}(\Sigma)$ of constants representing the elements of the domain of discourse; and D is a finite set of ground atoms expressed in terms of Σ.

Definition 2. Let $S = \langle \Sigma, D \rangle$ be a data-source and let P be a predicate that appears in D. Denote by P^S the set of tuples of P in D. We write $P(\overline{t}) \in P^S$, where $\overline{t}$ is a tuple of terms, to denote the formula $\bigvee_{\overline{a} \in P^S} (\overline{t} = \overline{a})$.

Example 2. Let $S = \langle \Sigma, D \rangle$ be the following data-source with facts about the relations $\mathsf{CarO}(\cdot,\cdot)$ (between people and their cars ID) and $\mathsf{Loc}(\cdot,\cdot)$ (between people and the place they live.)

$$\left\langle \left\{ \mathsf{CarO}/2, \mathsf{Loc}/2 \right\}, \left\{ \begin{array}{l} \mathsf{CarO}(\mathsf{JS}, \mathsf{V231}), \mathsf{CarO}(\mathsf{MC}, \mathsf{V231}), \mathsf{CarO}(\mathsf{MC}, \mathsf{B342}), \\ \mathsf{Loc}(\mathsf{JS}, \mathsf{Qn}), \mathsf{Loc}(\mathsf{MC}, \mathsf{Bx}) \end{array} \right\} \right\rangle .$$

Here, $\mathsf{Loc}^S = \{(\mathsf{JS}, \mathsf{Qn}), (\mathsf{MC}, \mathsf{Bx})\}$, hence $\mathsf{Loc}(x, y) \in \mathsf{Loc}^S$ denotes the following formula: $\big((x = \mathsf{JS}) \wedge (y = \mathsf{Qn})\big) \vee \big((x = \mathsf{MC}) \wedge (y = \mathsf{Bx})\big)$.

Standard mediator systems consist of a number of data-sources collaborating with information through a common interface, the global schema [8,14]. In such context, the data-sources can be viewed as storing information, in the form of a collection of tuples, about certain domain in the real world. However, which parts of the modeled world are accurately represented in the data-source is not recorded explicitly in the system, and so the *meaning* of the data-source remains ambiguous. With the following definition we address this problem by characterizing through a FOL expression -using the *same* language of the data-source- the cases in which the data-source contains all the valid facts. We call this the *window of expertise* of the data-source, and it is represented in the following definition by the formula Ψ.

Definition 3. A *local closed-world assumption* for a data-source $S = \langle \Sigma, D \rangle$, is a triple $\mathcal{LCWA} = \langle S, \overline{P}, \Psi \rangle$, where $\overline{P} = \{P_1(\overline{x}_1), \ldots, P_n(\overline{x}_n)\}$ is a set of atoms (the LCWA's objects) and $\Psi(\overline{y})$ (the context of the assumption) is a first-order formula over Σ with free variables $\overline{y}$ s.t. $\overline{y} \subseteq \bigcup_{i=1}^{n} \overline{x}_i$.

Note that in each $P_i(\overline{x}_i)$, the value of the variables $\overline{x}_i \cap \overline{y}$ are constrained by Ψ. For this reason we call Ψ the window of expertise, and $\exists \overline{y} \setminus \overline{x}_i(\Psi)$ the window of expertise of the predicate P_i. The intuitive meaning of the local closed-world assumption in Definition 3 is that for each $i \in \{1, \ldots, n\}$, each fact $P_i(\overline{x}_i)$ that is true in the real world and which satisfies $\exists \overline{y} \setminus \overline{x}_i(\Psi)$ should appear in the data-source.

Example 3. Let $S = \langle \Sigma, D \rangle$ the data-source of Example 2.

1. $\langle S, \{\mathsf{CarO}(x, y)\}, x = \mathsf{MC} \rangle$ intuitively indicates that the data-source S contains all true atoms of the form $\mathsf{CarO}(\mathsf{x}, \mathsf{y})$ for $x = \mathsf{MC}$.
2. $\langle S, \{\mathsf{CarO}(x, y)\}, \mathsf{Loc}(x, \mathsf{Bx}) \rangle$ indicates that S knows about all the cars of the people that live in Bx.

3. $\langle S, \{\mathsf{CarO}(x,y), \mathsf{Loc}(x,z)\}, \mathsf{Loc}(x, \mathsf{Bx})\rangle$ expresses that S contains all the data about the cars of persons living in Bx and about all people living in Bx.
4. $\langle S, \{\mathsf{CarO}(x,y), \mathsf{Loc}(x,z)\}, x = \mathsf{MC}\rangle$ indicates that S has *full knowledge* ab -out Mary Clark (i.e., a LCWA regarding everything that is concerned with MC).

Example 4. Consider the following two local closed-world assumptions:

$$\mathcal{LCWA}_A = \langle S, \{\mathsf{CarO}(x,y), \mathsf{Loc}(x,z)\}, \mathsf{CarO}(x, \mathsf{V231}) \wedge \mathsf{Loc}(x, \mathsf{Bx})\rangle$$
$$\mathcal{LCWA}_B = \langle S, \{\mathsf{CarO}(x,u), \mathsf{Loc}(y,v)\}, u = \mathsf{V231} \wedge v = \mathsf{Bx}\rangle$$

Intuitively, the difference between these two expressions is that the first one expresses a full knowledge of S about car ownership and locations of V231 owners in Bronx. On the other hand, under the second assumption, the data-source knows all people having a $V321$ including people not living in the Bronx; the data-source also knows all people living in the Bronx, including those that do not have a $V321$.

Example 5. In case of item (1) of Example 3, the local closed-world assumption may be expressed as follows:

$$\forall x\big(x = \mathsf{MC} \rightarrow \forall y(\mathsf{CarO}(x,y) \rightarrow y = \mathsf{V231} \vee y = \mathsf{B342})\big) \tag{1}$$

In case of item (2) of the same example, the local closed-world assumption may be expressed as follows:

$$\forall x\big(\mathsf{Loc}(x, \mathsf{Bx}) \rightarrow \forall y(\mathsf{CarO}(x,y) \rightarrow y = \mathsf{V231} \vee y = \mathsf{B342})\big) \tag{2}$$

These examples lead us to the following general formulation of a local closed-world assumption in terms of first-order formulae:

Definition 4. Let $\mathcal{LCWA} = \langle S, \{P_1(\overline{x}_1), \ldots, P_n(\overline{x}_n)\}, \Psi(\overline{y})\rangle$ be a local closed-world assumption for a data-source S. The formula that is *induced* from $\mathcal{LCWA}$, denoted by $\Lambda_{\mathcal{LCWA}}$, is the following:

$$\forall \overline{y}\Bigg(\Psi(\overline{y}) \rightarrow \forall \overline{z}\Big(\bigwedge_{i=1}^{n} \big(P_i(\overline{x}_i) \rightarrow \big(P_i(\overline{x}_i) \in P_i^D\big)\big)\Big)\Bigg)$$

where, $\overline{x} = \bigcup_{i=1}^{n} \overline{x}_i$, and $\overline{z} = \overline{x} \setminus \overline{y}$.

Note that if $\overline{P}$ is empty, then $\Lambda_{\mathcal{LCWA}}$ is tautologically true and does not specify any additional information.

Example 6. Below are, respectively, the formulae that are induced from the local closed-world assumptions of items (1) and (2) in Example 3.

1. $\forall x\big(x = \mathsf{MC} \rightarrow \forall y\big(\mathsf{CarO}(x,y) \rightarrow$
$((x = \mathsf{JS} \wedge y = \mathsf{V231}) \vee (x = \mathsf{MC} \wedge y = \mathsf{V231}) \vee (x = \mathsf{MC} \wedge y = \mathsf{B342}))\big)\big)$

2. $\forall x \big(\mathsf{Loc}(x, \mathsf{Bx}) \rightarrow \forall y (\mathsf{CarO}(x,y) \rightarrow$
$((x = \mathsf{JS} \wedge y = \mathsf{V231}) \vee (x = \mathsf{MC} \wedge y = \mathsf{V231}) \vee (x = \mathsf{MC} \wedge y = \mathsf{B342}))\ )\big)$

Note that under the unique name assumption (see Note 1 below), these formulae are equivalent with those of Example 5.[1]

Definition 5. For a data-source $S = \langle \Sigma, D \rangle$, denote: $\mathfrak{D}(S) = \bigwedge_{d \in D} d$.

Now we are ready to define the meaning of a data-source (in the context of mediator systems):

Definition 6. Let $S = \langle \Sigma, D \rangle$ be a data-source and let $\mathcal{LCWA}^j = \langle S, \overline{P}^j, \Psi^j \rangle$, $j = 1, \ldots, m$, be all the local closed-world assumptions of S. Then the *meaning* of S is given by the following formula:

$$\mathfrak{M}(S) = \mathfrak{D}(S) \wedge \bigwedge_{j=1}^{m} \Lambda_{\mathcal{LCWA}^j}.$$

Note 1. When $S = \langle \Sigma, D \rangle$ is the only data-source, the following two conditions are usually assumed:

– *Domain Closure Axiom*: $\mathsf{DCA}(S) = \forall x (\bigvee_{i=1}^{n} x = C_i)$
– *Unique Name Axiom*: $\mathsf{UNA}(S) = \bigwedge_{1 \leqslant i < j \leqslant n} C_i \neq C_j$

where $C_1, \ldots, C_n$ are all constants in Σ. In such cases, $\mathsf{DCA}(S)$ and $\mathsf{UNA}(S)$ appear as two additional conjuncts in $\mathfrak{M}(S)$. We denote the meaning of S by $\mathfrak{M}_D(S)$, $\mathfrak{M}_U(S)$, or $\mathfrak{M}_{DU}(S)$, when the first, the second or both assumptions are imposed, respectively.

The meaning of a data-source can be understood as a first-order theory representing incomplete knowledge about the real world. In the general case this theory will be incomplete, so there will exist more than one model, the actual world corresponding to one of them. Consequently, the meaning of a data-source is not be interpreted with respect to its database but with respect to the real world.

Given a formula Ψ, denote by $\exists|_{\overline{x}} \Psi$ the existential quantification of all free variables in Ψ, except those in $\overline{x}$.

The next proposition shows that the formula $\Lambda_{\mathcal{LCWA}}$ formalizes the intuitive meaning of the local closed-world assumption $\langle S, \overline{P}, \Psi \rangle$, as specified in the paragraph below Definition 3.

Proposition 1. *For $S = \langle \Sigma, D \rangle$, let $\mathcal{LCWA} = \langle S, \{P_1(\overline{x}_1), \ldots, P_n(\overline{x}_n)\}, \Psi \rangle$ and $\mathcal{LCWA}_i = \langle S, \{P_i(\overline{x_i})\}, \exists|_{\overline{x}_i} \Psi \rangle$ $i = 1, \ldots n$. Then:*

$$\Lambda_{\mathcal{LCWA}} \equiv \bigwedge_{i=1}^{n} \Lambda_{\mathcal{LCWA}_i}$$

[1] Consider, for instance, the first formula. It is of the form $\forall x \Phi$, and the formula in Example 3–(1) is of the form $\forall x \Phi'$. For every x other than MC both Φ and Φ' are trivially true, and for $x = \mathsf{MC}$, both Φ and Φ' hold only if there is no $c \notin \{\mathsf{V231}, \mathsf{B342}\}$ s.t. $\mathsf{CarO}(\mathsf{MC}, c)$ is true.

Proof. The equivalence is obtained by applying some simple rewriting rules on the relevant formulae. Indeed, denote $\overline{x} = \cup_{i=1}^{n} \overline{x}_i$ and $\overline{z} = \overline{x} \setminus \overline{y}$. Then:

$$\begin{aligned}
\Lambda_{\mathcal{LCWA}} &\equiv \forall \overline{y}\big(\Psi(\overline{y}) \rightarrow \big(\forall \overline{z}\big(\textstyle\bigwedge_{i=1}^{n} \big(P_i(\overline{x}_i) \rightarrow \big(P_i(\overline{x}_i) \in P_i^D\big)\big)\big)\big)\big) \\
&\equiv \forall \overline{y}\big(\Psi(\overline{y}) \rightarrow \big(\textstyle\bigwedge_{i=1}^{n} \forall \overline{z}\big(P_i(\overline{x}_i) \rightarrow \big(P_i(\overline{x}_i) \in P_i^D\big)\big)\big)\big) \\
&\equiv \forall \overline{y}\big(\textstyle\bigwedge_{i=1}^{n} \big(\Psi(\overline{y}) \rightarrow \forall(\overline{x}_i \setminus \overline{y})\big(P_i(\overline{x}_i) \rightarrow \big(P_i(\overline{x}_i) \in P_i^D\big)\big)\big) \\
&\equiv \textstyle\bigwedge_{i=1}^{n} \forall \overline{y}\big(\Psi(\overline{y}) \rightarrow \forall(\overline{x}_i \setminus \overline{y})\big(P_i(\overline{x}_i) \rightarrow \big(P_i(\overline{x}_i) \in P_i^D\big)\big)\big) \\
&\equiv \textstyle\bigwedge_{i=1}^{n} \forall(\overline{y} \cap \overline{x}_i)\big(\exists|_{\overline{x}}\Psi(\overline{y}) \rightarrow \forall(\overline{x_i} \setminus \overline{y})\big(P_i(\overline{x}_i) \rightarrow \big(P_i(\overline{x}_i) \in P_i^D\big)\big)\big) \\
&\equiv \textstyle\bigwedge_{i=1}^{n} \Lambda_{\mathcal{LCWA}_i}.
\end{aligned}$$

Thus the equivalence is obtained. □

Example 7. The assumption $\mathcal{LCWA} = \langle S, \{\mathsf{CarO}(x, y), \mathsf{Loc}(x, z)\}, x = \mathsf{MC}\rangle$, given in Example 3-(4), which says that S has full knowledge about Mary Clark, may also be represented in a modular way by the following two expressions:

$$\mathcal{LCWA}_A = \langle S, \{\mathsf{CarO}(x, y)\}, x = \mathsf{MC}\rangle$$
$$\mathcal{LCWA}_B = \langle S, \{\mathsf{Loc}(x, z)\}, x = \mathsf{MC}\rangle$$

We say that the meaning of a data-source is consistent if it has at least one model in the standard model-theoretic sense.

Proposition 2. *Every data-source has a consistent meaning.*

Proof. We consider the case where the meaning of a data-source $S = \langle \Sigma, D\rangle$ is given by $\mathfrak{M}(S)$. The proofs for $\mathfrak{M}_D(S)$, $\mathfrak{M}_U(S)$, and $\mathfrak{M}_{DU}(S)$ (i.e., when any combination of DCA and UNA is also assumed) are similar.

Let $\mathcal{LCWA}^j = \langle S, \overline{P}^j, \Psi^j\rangle$, $j=1,\ldots,m$ be all the local closed-world assumptions for S. Then $\mathfrak{M}(S) = \bigwedge_{A \in D} A \wedge \bigwedge_{j=1}^{m} \Lambda_{\mathcal{LCWA}^j}$. To show the proposition we define an interpretation I for Σ and show that it is a model of $\mathfrak{M}(S)$. Let I be the Herbrand interpretation associated with the database of S: the domain is $\mathcal{C}(\Sigma)$ and $P^I = P^S$. By construction of I, $I \models P_i(d_{i_1}, \ldots, d_{i_k})$ for every ground atom in D. When $\Psi^j(\overline{y}_j)$ is false in I, then trivially $I \models \bigwedge_{j=1}^{m} \Lambda_{\mathcal{LCWA}^j}$. When $\Psi^j(\overline{y}_j)$ true in I, then by its construction, whenever $I \models \overline{P}^j$, also $I \models P^j(\overline{x}) \in P^S$. □

The next proposition implies that for single data-sources our semantics of the local closed-world assumption is a conservative extension of Reiter's closed-world assumption for relational databases; the present approach allows to express in such cases that a single data-source has complete knowledge about the world.

Proposition 3. *Let $S = \langle \Sigma, D\rangle$ be the only data-source and let $\mathcal{LCWA} = \langle S, \overline{P}, \mathsf{TRUE}\rangle$, where $\overline{P} = \{P_1(\overline{x}_1), \ldots, P_n(\overline{x}_n)\}$ are all the predicates occurring in Σ. Then $\mathfrak{M}_{DU}(S)$ coincides with Reiter's axiomatization of the closed-world assumption* [13] *for S.*

Proof. Reiter's axiomatization of closed-world assumption of S is a first-order theory Γ, consisting of the following formulae: (1) $\mathsf{DCA}(S)$, (2) $\mathsf{UNA}(S)$, (3) the ground atomic facts in D, and (4) completion axioms for each predicate of S: $\forall\overline{x}(P_i(\overline{x}_i) \rightarrow P(\overline{x}_i) \in P_i^D)$, $i = 1, \ldots, n$.

By Definition 6, $\mathfrak{M}_{DU}(S)$ includes (1), (2) and (3), so it remains to show that $\mathcal{LCWA} = \langle S, \overline{P}, \mathsf{TRUE}\rangle$ is equivalent to (4). Indeed, the formula that is induced from this assumption is

$$\forall\overline{x}\Big(\bigwedge_{i=1}^{n} \big(P_i(\overline{x}_i) \rightarrow \big(P_i(\overline{x}_i) \in P_i^D\big)\big)\Big),$$

which is equivalent to the conjunction of the formulae in (4). □

3 A Circumscriptive Approach to the LCWA

In this section we consider an alternative approach to the representation of the closed-word assumption, this time by second-order formulas, and show its equivalence to the approach given in the previous section.

Consider again item 1 of Example 3. The local closed-world assumption in this case could be defined also in terms of sets as follows:

$$\{y \mid \mathsf{CarO}(\mathsf{MC}, y)\} = \{y \mid \mathsf{CarO}(\mathsf{MC}, y) \in D\}.$$

Since the set on the left-hand side of this equation is always a superset of the set on the right-hand side, the condition could be rephrased as follows:

$$\{y \mid \mathsf{CarO}(\mathsf{MC}, y)\} \subseteq \{y \mid \mathsf{CarO}(\mathsf{MC}, y) \in D\}.$$

This condition is specified in terms of a set inclusion property, and it is common to express such conditions by means of circumscriptive formulae. These formulae express the aspiration that the set of tuples of a certain predicate, satisfying a certain condition, should be as minimal as possible. It is not surprising, therefore, that a variant of the notion of local closed-world assumption presented here has already been expressed in term of circumscriptive axioms (see [2] and Section 5).

Definition 7. Let $\mathcal{LCWA} = \langle S, \{P_1(\overline{x}_1), \ldots, P_n(\overline{x}_n)\}, \Psi(\overline{y})\rangle$ be a local closed-world assumption for a data-source $S = \langle \Sigma, D\rangle$. The *pseudo-circumscriptive form* of $\mathcal{LCWA}$ is the following (second-order) formula, denoted $C(S)$:

$$\forall\overline{\Theta}\Bigg(\mathfrak{D}(S)[\overline{P}/\overline{\Theta}] \rightarrow \Bigg(\forall\overline{y}\Bigg(\Psi(\overline{y}) \rightarrow \forall\overline{z}(\overline{\Theta} \leqslant \overline{P})\Bigg) \rightarrow \forall\overline{y}\Bigg(\Psi(\overline{y}) \rightarrow \forall\overline{z}(\overline{P} \leqslant \overline{\Theta})\Bigg)\Bigg)\Bigg),$$

where $\overline{x} = \bigcup_{i=1}^{n} \overline{x}_i$, $\overline{z} = \overline{x} \setminus \overline{y}$, and

- $\overline{P} = \{P_1(\overline{x}_1), \ldots, P_n(\overline{x}_n)\}$, $\overline{\Theta} = \{\Theta_1(\overline{x}_1), \ldots, \Theta_n(\overline{x}_n)\}$, and each $\Theta_i(\overline{x}_i)$ is a predicate variable with the same arity of $P_i(\overline{x}_i)$,
- $\overline{P} \leqslant \overline{Q}$ is an abbreviation for $\bigwedge_{i=1}^{n}(P_i(\overline{x}_i) \rightarrow Q_i(\overline{x}_i))$. [2]

[2] $C(S)$ is called pseudo-circumscriptive since it differs from a pure circumscription schema by introducing the first-order formula Ψ into the representation. Just as in Definition 4, Ψ represents the context in which $\overline{P}$ should be minimal.

Definition 8. Let $S = \langle \Sigma, D \rangle$ be a data-source and let $C^j(S)$, $j = 1, \ldots, m$ be the pseudo-circumscriptive forms of its local closed-world assumptions. Denote:

$$\mathfrak{C}(S) = \mathfrak{D}(S) \wedge \bigwedge_{j=1}^{m} C^j(S).$$

Theorem 1. *For every data-source S, $\mathfrak{M}(S)$ is equivalent to $\mathfrak{C}(S)$.*

Proof. We prove the theorem for the case that $\overline{P}$ and $\overline{\Theta}$ are singletons, and that $m = 1$. The proof can be easily extended to the general case. We have to show that when $\mathfrak{D}(S)$ holds,

$$\forall \overline{y} \Big(\Psi(\overline{y}) \rightarrow \forall \overline{z} \big(P(\overline{x}) \rightarrow \big(P(\overline{x}) \in P^S \big) \big) \Big) \tag{3}$$

is equivalent to

$$\forall \Theta \Big(\underbrace{\mathfrak{D}(S)[P/\Theta]}_{(a)} \rightarrow \Big(\underbrace{\forall \overline{y} \Big(\Psi(\overline{y}) \rightarrow \forall \overline{z} \big(\Theta \leqslant P \big) \Big)}_{(b)} \rightarrow \underbrace{\forall \overline{y} \Big(\Psi(\overline{y}) \rightarrow \forall \overline{z} (P \leqslant \Theta) \Big)}_{(c)} \Big) \Big), \tag{4}$$

where, in both cases, $\overline{z} = \overline{x} \setminus \overline{y}$. Indeed,

($\Rightarrow$) Let I be a model of $\mathfrak{D}(S)$ and (3), and consider some value Θ^I in I for the predicate variable Θ. We show that if $\mathfrak{D}(S)[P/\Theta]$ is satisfied, so is the sub-formula (c) of (4), and hence the whole formula (4) is true as well. Let us prove, then, that sub-formula (c) holds. Assume that for some $\overline{y}$, $\Psi(\overline{y})$ is true in I and for some $\overline{z}$, $P(\overline{x})$ is true in I. As I is a model of (3), this implies that $P(\overline{x}) \in P^S$, i.e. for some tuple of terms $\overline{c}$ in the table of P in S, the equality $\overline{x} = \overline{c}^I$ holds in I. Since Θ^I satisfies $\mathfrak{D}(S)[P/\Theta]$, it follows that $\overline{x} \in \Theta^I$.

($\Leftarrow$) Let I be a model of $\mathfrak{D}(S)$ and (4). From $\mathfrak{D}(S)$ it follows that $\Theta \leqslant P$. It is obvious that $\Theta \leqslant P$ implies (b). Consequently (c) holds. Assume that there exist values $\overline{x}$ such that $\Psi(\overline{y})$ and $P(\overline{x})$ hold in I. To prove (3) we need to show that $P(\overline{x}) \in P^S$; or equivalently that there exists $\overline{c} \in P^S$ s.t. $\overline{x} = \overline{c}^I$. Because of (c) holds, it follows that $\overline{x} \in \Theta^I$. By our choice of Θ^I, this mean that for $\overline{c} \in P^S$, $\overline{x} = \overline{c}^I$. □

By the last theorem, the counterparts of Propositions 1, 2, and 3 in terms of $\mathfrak{C}(S)$ are also obtained.

Note 2. It is important to note that unless the data-sources consist of sets of facts, the first-order approach and the circumscriptive approach to the LCWA do *not* coincide. To see this, consider $S = \langle \{P/1\}, \{P(a) \vee P(b)\} \rangle$, and the assumption $\mathcal{LCWA} = \langle S, \{P(x)\}, \mathsf{TRUE} \rangle$. The formula in Definition 7 expresses a set inclusion minimization, and in this case it states an unconditional minimization of any extension of P. That is, an interpretation that satisfies both the disjunctive expression in S and the circumscriptive form of $\mathcal{LCWA}$, will necessarily state that either $P(a)$ or $P(b)$ is true, *but not both*. Intuitively, this can be read as "although the data-source is not complete with respect to P, at least it knows that no other element of the domain can belong to P, except of a or b (where the 'or' here is interpreted exclusively)".

4 Extensions and Additional Comments on the LCWA

4.1 LCWA with Several Data-Sources

An important (and intended) aspect of LCWA is applying it in a multiple-source environment. In this respect, it could be useful to specify a LCWA that addresses expertise obtained by the collective information in *several* data-sources. That is,

$$\mathcal{LCWA} = \langle \{S_1, \ldots, S_n\}, \overline{P}, \Psi \rangle. \tag{5}$$

should represent complete knowledge, shared by sources $\{S_1, \ldots, S_n\}$, in the context Ψ, about the predicates in $\overline{P}$. The induced formula $\Lambda_{\mathcal{LCWA}}$ of the assumption in (5) is obtained just as in the case of one data-source, when P^S is modified in the obvious way as follows:

Definition 9. Let $S_i = \langle \Sigma, D_i \rangle$, $i = 1, \ldots, n$ be n data-sources and let P be a predicate that appears in $\bigcup_{i=1}^{n} D_i$. Denote by $P^{\cup S_i}$ the set of tuples of P in $\bigcup_{i=1}^{n} D_i$, and abbreviate by $P(\overline{t}) \in P^{\cup S_i}$ the formula $\bigvee_{\overline{a} \in P^{\cup S_i}} (\overline{t} = \overline{a})$.

Now, the formula $\Lambda_{\mathcal{LCWA}}$ for the LCWA in (5) is defined just as the formula for one source, where $P(\overline{t}) \in P^S$ is replaced by $P(\overline{t}) \in P^{\cup S_i}$.

4.2 Complex Forms of LCWA

As local closed-world assumptions are first-order formulae, they can be used for expressing more complex assumptions about the information endorsed by the data-sources. For instance, a context (i.e., the third component) of one LCWA may be a formula that is induced by another LCWA, and so it is possible to 'compose' assumptions, and get, e.g., LCWA such as the following:

$$\mathcal{LCWA} = \Big\langle S_2, \{Q(\overline{x})\}, \Lambda_{\langle S_1\{P(\overline{x})\}, \mathsf{TRUE}\rangle} \Big\rangle \tag{6}$$

Note that the formula that is induced by assumption (6) is in fact equivalent to $\Lambda_{\langle S_1, \{P(\overline{x})\}, \mathsf{TRUE}\rangle} \rightarrow \Lambda_{\langle S_2, \{Q(\overline{x})\}, \mathsf{TRUE}\rangle}$, and in general,

$$\Lambda_{\langle S_2, \{Q(\overline{x})\}, \Lambda_{\langle S_1\{P(\overline{x})\}, \Psi\rangle}\rangle} = \Lambda_{\langle S_1, \{P(\overline{x})\}, \Psi\rangle} \rightarrow \Lambda_{\langle S_2, \{Q(\overline{x})\}, \mathsf{TRUE}\rangle}.$$

This idea also allows us to express more complicated assertions in terms of local closed-world assumptions. For instance, the following formula expresses that "either S_1 or S_2 has complete knowledge about P":

$$\big\{ \Lambda_{\langle S_1, \{P(\overline{x})\}, \mathsf{TRUE}\rangle} \vee \Lambda_{\langle S_2, \{P(\overline{x})\}, \mathsf{TRUE}\rangle} \big\}$$

Another possibility is to express that the assumptions about S_1 and S_2 are complementary, and so forth.

5 Related Works

The concept of a local closed-world assumption was first introduced in [4], in the context of knowledge bases for agents. The idea in that work was to represent a situation in which an agent has local closed-world information relative to a formula Φ and a knowledge base Γ, by a condition saying that every ground sentence that unifies with Φ either follows from Γ or is falsified by it. Formally:

$$\mathcal{LCWA}\ (\Phi) \equiv (\Gamma \models \Phi\theta) \vee (\Gamma \models \neg\Phi\theta) \text{ for all ground substitutions } \theta.$$

As we have noted above, a formal semantics for the definition of [4] in terms of second-order circumscription was proposed in [2]. The intuitive idea behind this semantics is the selection of only those models that satisfy the agent's knowledge-base and that are minimal with respect to the formulae for which the agent has complete information. We note, however, that the circumscriptive approach presented in [2] allows to minimize more predicates than those allowed by the pseudo-circumscriptive formula presented here. To see this consider, for instance $\mathcal{LCWA} = \langle S, \{P(x)\}, Q(x)\rangle$. Here, one may not know for which x, $Q(x)$ is true, and indeed the pseudo-circumscriptive formula of the $\mathcal{LCWA}$ does not affect $Q(x)$, but only $P(x)$ in the context of $Q(x)$. Suppose, then, that $P(a)$ is not in S, and we do not know whether $Q(a)$ is true, i.e. $Q(a)$ is not in S. In our approach, all we can derive is that if $Q(a)$ were true, then $P(a)$ would be false; but it is also possible that $P(a)$ is true but $Q(a)$ is false. Following the approach in [2], $P(x)$ and $Q(x)$ satisfy the data-source, but moreover, the intersection of $P(x)$ and $Q(x)$ should be minimal. In particular, if $P(b)$ is in S, but $Q(b)$ is not, then $Q(b)$ is considered false. So in this approach, also part of Q is minimized, not only P.

An alternative approach to express different levels of knowledge of a certain data-source with respect to the global domain is to label the predicates of the data-sources as "sound", "complete" or "exact" (see, for instance, [1,5,6]). We identify two main drawbacks with this approach. The first one is the loss of elegance and flexibility by the introduction of non-logical symbols to the representation. The second, more serious problem, is related to the limitation in grasping more refined knowledge about the specific areas in which the predicates of the data-source contain complete information, as observed in several examples in this paper.

In [12], the concepts of "coverage" and "density" were introduced in order to measure the completeness of data-sources at the intensional and extensional levels, respectively. The authors use these concepts to determine the completeness of one or more data-sources, gathered under merge operators. As in our approach, the intension and the contents of the predicates in a data-source are divided into two independent components. This allows to provide a general completeness measure for the data-sources, but again, it is not possible to explicitly specify situations in which the data-sources have complete knowledge about (parts of) the domain of discourse.

6 Conclusion and Future Work

In this paper we presented a method of expressing the meaning of a data-source in the context of information systems that mediate among several sources. A key issue in this respect is the ability to properly define and represent particular cases where there is a complete knowledge, although partial knowledge of the sources is usually assumed. The resulting theory is expressed by a first-order one. It may also be represented by circumscriptive-like formulae.

This is an ongoing work which is part of a larger project aiming to represent and reason with incomplete information in general mediator-based systems. In such broader context a number of related issues should be addressed as well. Below we consider some of them.

- Expressing meta-knowledge about the data-sources themselves. For instance, while it is possible to express by our approach statements such as "the data-source S contains complete knowledge about car owners in Bronx", it is not possible to represent a statement such as "for every car in Bronx that is known to the data-source S, S also knows its owners". While the first statement refers to the knowledge that S possesses about the domain of discourse, the latter expresses knowledge about S itself. In order to represent the second kind of statements, an extension based on modalities in the spirit of [7] seems to be a natural candidate.
- Consider the assumption $\mathcal{LCWA}^* = \langle S, \{P(x)\}, \neg Q(x) \rangle$. If no other assumption mentions Q in its second component, this assertion does not allow to conclude whether S has complete knowledge about P. Indeed, the induced formula in this case is of the form $\Lambda_{\mathcal{LCWA}} = \forall x. \neg Q(x) \rightarrow \ldots$, but the validity of $\neg Q(x)$ cannot be verified, since the data-sources mention only positive information. Of course, if there are other assumptions, for instance, $\mathcal{LCWA}^{**} = \langle S, \{Q(x)\}, \mathsf{TRUE} \rangle$ (which implies complete knowledge about Q) then $\mathcal{LCWA}^*$ states that for all x such that $Q(x)$ is not in the database, if $P(x)$ is true then S contains $P(x)$. This situation shows that in order to obtain complete knowledge about an arbitrary predicate P under its window of expertise, the formula Ψ must define unambiguously such window. The specific conditions for which Ψ define complete knowledge over source predicates is a crucial issue that must be investigated in the depth, since it would allow to discriminate from a set of LCWA expressions which ones are useful in practice.
- While this paper concentrates on representation forms of the closed-world-assumption and their properties, computational aspects of reasoning with these assumptions should be considered as well. Among the issues that should be addressed is the effect of the local closed-world assumptions on the complexity and decidability of the resulting theories.
- Finding a proper way to incorporate the information that the mediator system has about its data-sources with the theory that relates the different terminologies of the data-sources and the global vocabulary (called *schema mappings*). This information may also be used for splitting global queries among the sources to obtain sound and complete answers (*query planning*).

References

1. D. Calvanese, G. De Giacomo, and M. Lenzerini. Description logics for information integration. In *Computational Logic: Logic Programming and Beyond, Essays in Honour of Robert A. Kowalski, Part II*, LNCS 2408, pages 41–60. 2002.
2. P. Doherty, W. Łukaszewicz, and A. Szalas. Efficient reasoning using the local closed-world assumption. In *Proc. 9th AIMSA*, LNCS 2407, pages 49–58, 2000.
3. O Duschka, M. Genesereth, and A. Levy. Recursive query plans for data integration. *J. Logic Programming*, 43(1):49–73, 2000.
4. O. Etzioni, K. Golden, and D. Weld. Sound and efficient closed-world reasoning for planning. *Artificial Intelligence*, 89(1-2):113–148, 1997.
5. G. Grahne. Information integration and incomplete information. *IEEE Data Engineering Bulletin*, 25(3):46–52, 2002.
6. G. Grahne and A. Mendelzon. Tableau techniques for querying information sources through global schemas. In *Proc. 7th ICDT*, LNCS 1540, pages 332–347, 1999.
7. J. Halpern and Y. Moses. Knowledge and common knowledge in a distributed environment. *Journal of the ACM*, 37(3):549–587, 1990.
8. M. Lenzerini. Data integration: A theoretical perspective. In *Proc. 21st PODS*, pages 233–246, 2002.
9. Rousset M. and Reynaud Ch. Knowledge representation for information integration. *Inf. Syst.*, 29(1):3–22, 2004.
10. J. McCarthy. Applications of circumscription to formalizing common sense knowledge. In V. Lifschitz, editor, *Formalizing Common Sense: Papers by John McCarthy*, pages 198–225. Ablex Publishing Corporation, 1990.
11. T. Millstein, A. Levy, and M. Friedman. Query containment for data integration systems. In *Proc. 21st PODS*, pages 67–75, 2002.
12. F. Naumann, J.C. Freytag, and U. Leser. Completeness of integrated information sources. *Information Systems*, 29(7):583–615, 2004.
13. R. Reiter. Towards a logical reconstruction of relational database theory. In *On Conceptual Modelling, Perspectives from Artificial Intelligence, Databases, and Programming Languages.*, pages 191–233, 1982.
14. B. Van Nuffelen, A. Cortés-Calabuig, M. Denecker, O. Arieli, and M. Bruynooghe. Data integration using ID-logic. In *Proc. 16th CAiSE*, LNCS 3084, pages 67–81, 2004.

Computing Dialectical Trees Efficiently in Possibilistic Defeasible Logic Programming

Carlos I. Chesñevar[1], Guillermo R. Simari[2], and Lluis Godo[3]

[1] Departament of Computer Science, Universitat de Lleida,
C/Jaume II, 69, 25001 Lleida, Spain
cic@eps.udl.es

[2] Department of Computer Science and Engineering, Universidad Nacional del Sur,
Alem 1253, (8000) Bahía Blanca, Argentina
grs@cs.uns.edu.ar

[3] Artificial Intelligence Research Institute (IIIA-CSIC),
Campus UAB, 08193 Bellaterra, Barcelona Spain
godo@iiia.csic.es

Abstract. Possibilistic Defeasible Logic Programming (P-DeLP) is a logic programming language which combines features from argumentation theory and logic programming, incorporating as well the treatment of possibilistic uncertainty and fuzzy knowledge at object-language level. Solving a P-DeLP query Q accounts for performing an exhaustive analysis of arguments and defeaters for Q, resulting in a so-called dialectical tree, usually computed in a depth-first fashion. Computing dialectical trees efficiently in P-DeLP is an important issue, as some dialectical trees may be computationally more expensive than others which lead to equivalent results. In this paper we explore different aspects concerning how to speed up dialectical inference in P-DeLP. We introduce definitions which allow to characterize dialectical trees constructively rather than declaratively, identifying relevant features for pruning the associated search space. The resulting approach can be easily generalized to be applied in other argumentation frameworks based in logic programming.

Keywords: Defeasible Argumentation, Logic Programming, Dialectical Reasoning.

1 Introduction and Motivations

Possibilistic Defeasible Logic Programming (P-DeLP) [1] is a logic programming language which combines features from argumentation theory and logic programming, incorporating as well the treatment of possibilistic uncertainty and fuzzy knowledge at object-language level. As in many argumentation frameworks based in logic programming, solving a P-DeLP query Q accounts for performing an exhaustive analysis of arguments and defeaters for Q, resulting in a so-called dialectical tree, usually computed in a depth-first fashion.

Computing dialectical trees efficiently in P-DeLP is an important issue, as some dialectical trees may be computationally more expensive than others which

C. Baral et al. (Eds.): LPNMR 2005, LNAI 3662, pp. 158–171, 2005.
© Springer-Verlag Berlin Heidelberg 2005

lead to equivalent results. In this paper we explore different aspects concerning how to speed up dialectical inference in P-DeLP. We introduce definitions which allow to characterize dialectical trees constructively rather than declaratively, identifying relevant features for pruning the associated search space. The resulting approach can be easily generalized to be applied in other argumentation frameworks based in logic programming.

The rest of the paper is structured as follows. Section 2 summarizes the details of P-DeLP. Section 3 discusses how computation of dialectical trees can be modelled in the context of P-DeLP, a characterization extensible to other similar frameworks. Section 4 presents a generic algorithm for computing dialectical trees in a depth-first fashion, as well as some criteria to be considered for pruning the resulting search space. Finally, Section 5 summarizes related work and the main conclusions that have been obtained.

2 The P-DeLP Programming Language: Fundamentals

The P-DeLP language $\mathcal{L}$ is defined from a set of ground fuzzy atoms (fuzzy propositional variables) $\{p, q, \ldots\}$ together with the connectives $\{\sim, \wedge, \leftarrow\}$. The symbol $\sim$ stands for *negation*. A *literal* $L \in \mathcal{L}$ is a ground (fuzzy) atom q or a negated ground (fuzzy) atom $\sim q$, where q is a ground (fuzzy) propositional variable. A *rule* in $\mathcal{L}$ is a formula of the form $Q \leftarrow L_1 \wedge \ldots \wedge L_n$, where $Q, L_1, \ldots, L_n$ are literals in $\mathcal{L}$. When $n = 0$, the formula $Q \leftarrow$ is called a *fact* and simply written as Q. The term *goal* will be used to refer to any literal $Q \in \mathcal{L}$.[1] In the following, capital and lower case letters will denote literals and atoms in $\mathcal{L}$, respectively.

Definition 1 (P-DeLP formulas). *The set Wffs($\mathcal{L}$) of wffs in $\mathcal{L}$ are* facts, rules *and* goals *built over the literals of $\mathcal{L}$. A* certainty-weighted *clause in $\mathcal{L}$, or simply* weighted clause*, is a pair of the form (φ, α), where $\varphi \in$ Wffs($\mathcal{L}$) and $\alpha \in [0, 1]$ expresses a lower bound for the certainty of φ in terms of a necessity measure.*

The original P-DeLP language [1] is based on Possibilistic Gödel Logic or PGL [2], which is able to model both uncertainty and fuzziness and allows for a partial matching mechanism between fuzzy propositional variables. For simplicity and space reasons we will restrict ourselves to fragment of P-DeLP built on non-fuzzy propositions, and hence based on the necessity-valued classical propositional Possibilistic logic [3]. As a consequence, possibilistic models are defined by possibility distributions on the set of classical interpretations [2] and the proof method for our P-DeLP formulas, written $\vdash$, is defined based on the following generalized modus ponens rule (GMP):

---

[1] Note that conjunction of literals is not a valid goal.

[2] Although the connective $\leftarrow$ in logic programming is different form the material implication, e.g. $p \leftarrow q$ is not the same as $\sim q \leftarrow \sim p$, regarding the possibilistic semantics we assume here they share the same set interpretations.

$$\frac{\begin{array}{c}(L_0 \leftarrow L_1 \wedge \cdots \wedge L_k, \gamma)\\ (L_1, \beta_1), \ldots, (L_k, \beta_k)\end{array}}{(L_0, \min(\gamma, \beta_1, \ldots, \beta_k))}$$

which is a particular instance of the well-known possibilistic resolution rule, and which provides the *non-fuzzy* fragment of P-DeLP with a complete calculus for determining the maximum degree of possibilistic entailment for weighted literals.[3]

In P-DeLP we distinguish between *certain* and *uncertain* clauses. A clause (φ, α) will be referred as certain if $\alpha = 1$ and uncertain, otherwise. Moreover, a set of clauses Γ will be deemed as *contradictory*, denoted $\Gamma \vdash \perp$, if $\Gamma \vdash (q, \alpha)$ and $\Gamma \vdash (\sim q, \beta)$, with $\alpha > 0$ and $\beta > 0$, for some atom q in $\mathcal{L}$.[4] A P-DeLP program is a set of weighted rules and facts in $\mathcal{L}$ in which we distinguish certain from uncertain information. As additional requirement, certain knowledge is required to be non-contradictory. Formally:

Definition 2 (Program). *A P-DeLP program $\mathcal{P}$ (or just program $\mathcal{P}$) is a pair (Π, Δ), where Π is a non-contradictory finite set of certain clauses, and Δ is a finite set of uncertain clauses.*

Example 1. Consider an intelligent agent controlling an engine with three switches $sw1$, $sw2$ and $sw3$. These switches regulate different features of the engine, such as pumping system, speed, etc. The knowledge of such an agent can be modelled by the program $\mathcal{P}_{eng}$ shown in Fig. 1. Note that uncertainty is assessed in terms of different necessity measures. This agent may have the following certain and uncertain knowledge about how this engine works, *e.g.* *"if the pump is clogged, then the engine gets no fuel with necessity measure of 1"* (rule 1) or *"When there is heat, then oil is usually not ok with necessity measure of 0.9"* (rule 12). Suppose also that the agent knows that $sw1$, $sw2$ and $sw3$ are on, and there is *heat* (rules 1-5). The agent wants to determine if the engine is ok on the basis of this program $\mathcal{P}_{eng}$.

(1)	$(\sim fuel_ok \leftarrow pump_clog, 1)$	(9)	$(oil_ok \leftarrow pump_oil, 0.8)$
(2)	$(sw1, 1)$	(10)	$(engine_ok \leftarrow fuel_ok \wedge oil_ok, 0.3)$
(3)	$(sw2, 1)$	(11)	$(\sim engine_ok \leftarrow fuel_ok \wedge oil_ok \wedge heat, 0.95)$
(4)	$(sw3, 1)$	(12)	$(\sim oil_ok \leftarrow heat, 0.9)$
(5)	$(heat, 1)$	(13)	$(pump_clog \leftarrow pump_fuel \wedge low_speed, 0.7)$
(6)	$(pump_fuel \leftarrow sw1, 0.6)$	(14)	$(low_speed \leftarrow sw2, 0.8)$
(7)	$(fuel_ok \leftarrow pump_fuel, 0.3)$	(15)	$(\sim low_speed \leftarrow sw2, sw3, 0.8)$
(8)	$(pump_oil \leftarrow sw2, 0.8)$	(16)	$(fuel_ok \leftarrow sw3, 0.9)$

Fig. 1. P-DeLP program $\mathcal{P}_{eng}$ (example 1)

[3] From now on we will simply use P-DeLP to actually refer to the non-fuzzy fragment of the original P-DeLP.

[4] For a given goal Q, we write $\sim Q$ as an abbreviation to denote "$\sim q$" if $Q \equiv q$ and "q" if $Q \equiv \sim q$.

Definition 3 (Argument. Subargument). *Given a program* $\mathcal{P} = (\Pi, \Delta)$*, a set* $\mathcal{A} \subseteq \Delta$ *of uncertain clauses is an* argument *for a goal* Q *with necessity degree* $\alpha > 0$*, denoted* $\langle \mathcal{A}, Q, \alpha \rangle$*, iff: (1)* $\Pi \cup \mathcal{A} \vdash (Q, \alpha)$*; (2)* $\Pi \cup \mathcal{A}$ *is non contradictory; and (3) There is no* $\mathcal{A}_1 \subset \mathcal{A}$ *such that* $\Pi \cup \mathcal{A}_1 \vdash (Q, \beta)$*,* $\beta > 0$*. Let* $\langle \mathcal{A}, Q, \alpha \rangle$ *and* $\langle \mathcal{S}, R, \beta \rangle$ *be two arguments. We will say that* $\langle \mathcal{S}, R, \beta \rangle$ *is a* subargument *of* $\langle \mathcal{A}, Q, \alpha \rangle$ *iff* $\mathcal{S} \subseteq \mathcal{A}$*. Notice that the goal* R *may be a subgoal associated with the goal* Q *in the argument* $\mathcal{A}$*.*

Note that from the definition of argument, it follows that on the basis of a P-DeLP program $\mathcal{P}$ there may exist different arguments $\langle \mathcal{A}_1, Q, \alpha_1 \rangle$, $\langle \mathcal{A}_2, Q, \alpha_2 \rangle$, $\ldots$, $\langle \mathcal{A}_k, Q, \alpha_k \rangle$ supporting a given goal Q, with (possibly) different necessity degrees $\alpha_1, \alpha_2, \ldots, \alpha_k$. Arguments are built by backward chaining on the basis of the P-DeLP program P. The necessity degree of the conclusion of an argument involving clauses $(C_1, \beta_1), \ldots (C_k, \beta_k)$ is defined as $min(\beta_1, \ldots, \beta_k)$. Consequently, if $\langle \mathcal{S}, R, \beta \rangle$ is a subargument of an argument $\langle \mathcal{A}, Q, \alpha \rangle$, then $\beta \geq \alpha$.

Example 2. Consider theprogram $\mathcal{P}_{eng}$ in Ex 1. The argument $\langle \mathcal{A}_1, engine_ok, 0.3 \rangle$ can be obtained, with
$\mathcal{A}_1 = \{(engine_ok \leftarrow fuel_ok \wedge oil_ok, 0.3), (pump_fuel \leftarrow sw1, 0.6);$
$(fuel_ok \leftarrow pump_fuel, 0.3), \{(pump_oil \leftarrow sw2, 0.8);(oil_ok \leftarrow pump_oil, 0.8)\}.$
In particular, the argument $\langle \mathcal{B}, fuel_ok, 0.3 \rangle$, with $\mathcal{B}= \{(pump_fuel \leftarrow sw1, 0.6);$ $(fuel_ok \leftarrow pump_fuel, 0.3)\}$, is a subargument of $\langle \mathcal{A}_1, engine_ok, 0.3 \rangle$.

Conflict among arguments will be formalized by the notions of counterargument and defeat presented next.

Definition 4 (Counterargument). *Let* $\mathcal{P}$ *be a program, and let* $\langle \mathcal{A}_1, Q_1, \alpha_1 \rangle$ *and* $\langle \mathcal{A}_2, Q_2, \alpha_2 \rangle$ *be two arguments wrt* $\mathcal{P}$*. We will say that* $\langle \mathcal{A}_1, Q_1, \alpha_1 \rangle$ *counterargues* $\langle \mathcal{A}_2, Q_2, \alpha_2 \rangle$ *iff there exists a subargument (called* disagreement subargument*)* $\langle \mathcal{S}, Q, \beta \rangle$ *of* $\langle \mathcal{A}_2, Q_2, \alpha_2 \rangle$ *such that* $\Pi \cup \{(Q_1, \alpha_1), (Q, \beta)\}$ *is contradictory. The literal* (Q, β) *will be called* disagreement literal.

Example 3. Consider the program from Ex 1. Another argument $\langle \mathcal{A}_2, \sim fuel_ok, 0.6 \rangle$ can be found, with

$$\mathcal{A}_2 = \{ (\sim fuel_ok \leftarrow sw1, 0.6), (low_speed \leftarrow sw2, 0.8), (pump_clog \leftarrow pump_fuel \wedge low_speed, 0.7)\}$$

Argument $\langle \mathcal{A}_2, \sim fuel_ok, 0.6 \rangle$ is a counterargument for $\langle \mathcal{A}_1, engine_ok, 0.3 \rangle$ as there exists a subargument $\langle \mathcal{B}, fuel_ok, 0.3 \rangle$ in $\langle \mathcal{A}_1, engine_ok, 0.3 \rangle$ (see Example 2) such that $\Pi \cup \{(fuel_ok, 0.3), (\sim fuel_ok, 0.6)\}$ is contradictory.

Defeat among arguments involves a *preference criterion* on conflicting arguments, defined on the basis of necessity measures associated with arguments.

Definition 5 (Defeat). *Let* $\mathcal{P}$ *be a program, and let* $\langle \mathcal{A}_1, Q_1, \alpha_1 \rangle$ *and* $\langle \mathcal{A}_2, Q_2, \alpha_2 \rangle$ *be two arguments in* $\mathcal{P}$*. We will say that* $\langle \mathcal{A}_1, Q_1, \alpha_1 \rangle$ *is a* defeater *for* $\langle \mathcal{A}_2, Q_2, \alpha_2 \rangle$ *iff* $\langle \mathcal{A}_1, Q_1, \alpha_1 \rangle$ *counterargues argument* $\langle \mathcal{A}_2, Q_2, \alpha_2 \rangle$ *with disagreement subargument* $\langle \mathcal{A}, Q, \alpha \rangle$*, with* $\alpha_1 \geq \alpha$*. If* $\alpha_1 > \alpha$ *then* $\langle \mathcal{A}_1, Q_1, \alpha_1 \rangle$ *is called a* proper defeater*, otherwise (*$\alpha_1 = \alpha$*) it is called a* blocking defeater.

Example 4. Consider $\langle \mathcal{A}_1, engine_ok, 0.3 \rangle$ and $\langle \mathcal{A}_2, \sim fuel_ok, 0.6 \rangle$ in Ex. 3. Then $\langle \mathcal{A}_2, \sim fuel_ok, 0.6 \rangle$ is a proper defeater for $\langle \mathcal{A}_1, engine_ok, 0.3 \rangle$, as $\langle \mathcal{A}_2, \sim fuel_ok, 0.6 \rangle$ counterargues $\langle \mathcal{A}_1, engine_ok, 0.3 \rangle$ with disagreement subargument $\langle \mathcal{B}, fuel_ok, 0.3 \rangle$, and $0.6 > 0.3$.

Definition 6 (Argumentation line). *An* argumentation line λ *starting in an argument* $\langle \mathcal{A}_0, Q_0, \alpha_0 \rangle$ *is a finite sequence of arguments* $[\langle \mathcal{A}_0, Q_0, \alpha_0 \rangle, \langle \mathcal{A}_1, Q_1, \alpha_1 \rangle, \ldots, \langle \mathcal{A}_n, Q_n, \alpha_n \rangle, \ldots]$ *such that every* $\langle \mathcal{A}_i, Q_i, \alpha_i \rangle$ *defeats* $\langle \mathcal{A}_{i-1}, Q_{i-1}, \alpha_{i-1} \rangle$*, for* $0 < i \leq n$*, satisfying certain* dialectical constraints *(see below). Every argument* $\langle \mathcal{A}_i, Q_i, \alpha_i \rangle$ *in* λ *has* level i*. We will distinguish the sets*

$$S^k_\lambda = \bigcup_{i=0,2,\ldots,2\lfloor k/2 \rfloor} \{\langle \mathcal{A}_i, Q_i, \alpha_i \rangle \in \lambda\} \text{ and } I^k_\lambda = \bigcup_{i=1,3,\ldots,2\lfloor k/2 \rfloor+1} \{\langle \mathcal{A}_i, Q_i, \alpha_i \rangle \in \lambda\}$$

associated with even-level (resp. odd-level) arguments in λ *up to the* k*-th level* $(k \leq n)$*.*

An argumentation line can be thought of as an exchange of arguments between two parties, a *proponent* (evenly-indexed arguments) and an *opponent* (oddly-indexed arguments). In order to avoid *fallacious* reasoning, argumentation theory imposes additional constraints on such an argument exchange to be considered rationally acceptable wrt a P-DeLP program $\mathcal{P}$, namely:[5]

1. **Non-contradiction:** given an argumentation line λ of length n the set S^n_λ associated with the proponent (resp. I^n_λ for the opponent) should be *non-contradictory* wrt $\mathcal{P}$.[6]
2. **No circular argumentation:** no argument $\langle \mathcal{A}_j, Q_j, \alpha_j \rangle$ in λ is a sub-argument of an argument $\langle \mathcal{A}_i, Q_i, \alpha_i \rangle$ in λ, $i < j$.
3. **Progressive argumentation:** every blocking defeater $\langle \mathcal{A}_i, Q_i, \alpha_i \rangle$ in λ is defeated by a proper defeater $\langle \mathcal{A}_{i+1}, Q_{i+1}, \alpha_{i+1} \rangle$ in λ.

An argumentation line that cannot be further extended on the basis of a given program $\mathcal{P}$ will be called *exhaustive*, otherwise it will be *partial*. Formally:

Definition 7 (Partial/exhaustive argumentation line). *Given two argumentation lines* λ *and* λ'*, we will say that* λ' extends λ *iff* λ *is an initial subsequence of* λ'*. An argumentation line* λ *will be called* exhaustive *iff there is no argumentation line* λ' *that extends* λ*; otherwise* λ *will be called* partial.

As most argumentation systems [5,6], in order to determine whether a given argument is ultimately undefeated (or *warranted*) wrt a program $\mathcal{P}$, the P-DeLP framework relies on an *exhaustive* dialectical analysis. Such analysis is modelled in terms of a *dialectical tree*,[7] where every path can be seen as an exhaustive argumentation line.

Definition 8 (Dialectical tree). *Let* $\mathcal{P}$ *be a program, and let* $\langle \mathcal{A}_0, Q_0, \alpha_0 \rangle$ *be an argument wrt* $\mathcal{P}$*. A* dialectical tree *for* $\langle \mathcal{A}_0, Q_0, \alpha_0 \rangle$*, denoted* $\mathcal{T}_{\langle \mathcal{A}_0, Q_0, \alpha_0 \rangle}$*, is a tree structure defined as follows:*

[5] These constraints may vary from one particular argumentation framework to another. In particular, parametrizing dialectical trees with constraints on argumentation lines may give rise to characterizations of different logic programming semantics, as shown in [4].

[6] Non-contradiction for a set of arguments is defined as a generalization of Def. 3: a set $S = \bigcup_{i=1}^n \{\langle \mathcal{A}_i, Q_i, \alpha_i \rangle\}$ of arguments is *contradictory* wrt $\mathcal{P}$ iff $\Pi \cup \bigcup_{i=1}^n \mathcal{A}_i$ is contradictory.

[7] In some frameworks other names are used for denoting tree-like structures of arguments, *e.g.* 'argument tree' or 'dialogue tree' [7,8].

1. *The root node of* $\mathcal{T}_{\langle \mathcal{A}_0, Q_0, \alpha_0 \rangle}$ *is* $\langle \mathcal{A}_0, Q_0, \alpha_0 \rangle$.
2. $\langle \mathcal{B}', H', \beta' \rangle$ *is an immediate child of* $\langle \mathcal{B}, H, \beta \rangle$ *iff there exists an* ***exhaustive*** *argumentation line* $\lambda = [\langle \mathcal{A}_0, Q_0, \alpha_0 \rangle, \langle \mathcal{A}_1, Q_1, \alpha_1 \rangle, \ldots, \langle \mathcal{A}_n, Q_n, \alpha_n \rangle, \ldots]$ *such that there are two elements* $\langle \mathcal{A}_{i+1}, Q_{i+1}, \alpha_{i+1} \rangle = \langle \mathcal{B}', H', \beta' \rangle$ *and* $\langle \mathcal{A}_i, Q_i, \alpha_i \rangle = \langle \mathcal{B}, H, \beta \rangle$*, for some* $i = 0 \ldots n-1$.

Example 5. Consider Ex. 1. To compute the dialectical tree for $\langle \mathcal{A}_1, engine_ok, 0.3 \rangle$, the P-DeLP inference engine computes argumentation lines by depth-first search. A defeater for $\langle \mathcal{A}_1, engine_ok, 0.3 \rangle$ is found, namely $\langle \mathcal{A}_2, \sim fuel_ok, 0.6 \rangle$ (Ex. 4). This defeater can on its turn be defeated by a third defeater $\langle \mathcal{A}_3, \sim low_speed, 0.8 \rangle$, with disagreement subargument $\langle \mathcal{A}_2', low_speed, 0.8 \rangle$. Note that argument $\langle \mathcal{A}_4, fuel_ok, 0.9 \rangle$, with $\mathcal{A}_4 = \{\ (fuel_ok \leftarrow sw3, 0.9)\ \}$ would be also a defeater for $\langle \mathcal{A}_2, \sim fuel_ok, 0.6 \rangle$. This completes the analysis of defeaters for $\langle \mathcal{A}_2, \sim fuel_ok, 0.6 \rangle$. Backtracking to argument $\langle \mathcal{A}_1, engine_ok, 0.3 \rangle$, another defeater is found, namely $\langle \mathcal{A}_5, \sim engine_ok, 0.3 \rangle$, with

$$\mathcal{A}_5 = \{\ (\sim engine_ok \leftarrow fuel_ok \wedge oil_ok \wedge heat, 0.95)\ ;\ (pump_fuel \leftarrow sw1, 0.6); \\ (fuel_ok \leftarrow pump_fuel, 0.3),\ (pump_oil \leftarrow sw2, 0.8);\ (oil_ok \leftarrow pump_oil, 0.8)\}.$$

Note that no further arguments can be found in the dialectical analysis. Although $\langle \mathcal{A}_6, \sim oil_ok, 0.9 \rangle$, with $\mathcal{A}_6 = \{\ (\sim oil_ok \leftarrow heat, 0.9)\ \}$ seems a possible defeater for $\langle \mathcal{A}_5, \sim engine_ok, 0.3 \rangle$, such argument would be *fallacious* as $\mathcal{A}_1$ supports *oil_ok*, and there would be even-level arguments (namely $\mathcal{A}_1$ and $\mathcal{A}_5$) supporting contradictory conclusions. Therefore three different exhaustive argumentation lines can be computed, namely rooted in $\langle \mathcal{A}_1, engine_ok, 0.3 \rangle$, namely:

- $[\langle \mathcal{A}_1, engine_ok, 0.3 \rangle, \langle \mathcal{A}_2, \sim fuel_ok, 0.6 \rangle, \langle \mathcal{A}_3, \sim low_speed, 0.8 \rangle\ ]$
- $[\ \langle \mathcal{A}_1, engine_ok, 0.3 \rangle, \langle \mathcal{A}_2, \sim fuel_ok, 0.6 \rangle, \langle \mathcal{A}_4, fuel_ok, 0.9 \rangle\ ]$
- $[\langle \mathcal{A}_1, engine_ok, 0.3 \rangle, \langle \mathcal{A}_5, \sim engine_ok, 0.3 \rangle\ ]$

Fig. 2(b) shows the corresponding dialectical tree $\mathcal{T}_{\langle \mathcal{A}_1, engine_ok, 0.3 \rangle}$.

Nodes in a dialectical tree $\mathcal{T}_{\langle \mathcal{A}_0, Q_0, \alpha_0 \rangle}$ can be marked as *undefeated* and *defeated* nodes (U-nodes and D-nodes, resp.). A dialectical tree will be marked as an AND-OR tree: all leaves in $\mathcal{T}_{\langle \mathcal{A}_0, Q_0, \alpha_0 \rangle}$ will be marked U-nodes (as they have no defeaters), and every inner node is to be marked as *D-node* iff it has at least one U-node as a child, and as *U-node* otherwise. Note that $\alpha - \beta$ pruning (see Fig. 2(a)) can be applied, so not every node in the tree needs to be generated. We will write $Mark(\mathcal{T}_i) = U$ (resp.$Mark(\mathcal{T}_i) = D$) to denote that the root node of $\mathcal{T}_i$ is marked as U-node (resp. D-node).

Definition 9 (Warrant). *An argument* $\langle \mathcal{A}_0, Q_0, \alpha_0 \rangle$ *is ultimately accepted as valid (or* warranted*) with a necessity degree* α_0 *wrt a program* $\mathcal{P}$ *iff the root of the tree* $\mathcal{T}_{\langle \mathcal{A}_0, Q_0, \alpha_0 \rangle}$ *is marked as* U-node *(i.e.,* $Mark(\mathcal{T}_{\langle \mathcal{A}_0, Q_0, \alpha_0 \rangle}) = U$*).*

Example 6. Consider $\mathcal{T}_{\langle \mathcal{A}_1, engine_ok, 0.3 \rangle}$ in Ex. 5. Fig. 2(b) shows the result of computing $Mark(\mathcal{T}_{\langle \mathcal{A}_1, engine_ok, 0.3 \rangle}) = D$ with $\alpha - \beta$ pruning. From Def. 9 it holds that $\langle \mathcal{A}_1, engine_ok, 0.3 \rangle$ is not warranted.

3 Modelling the Computation of Dialectical Trees

P-DeLP –as well as other implemented logic programming approaches to argumentation– relies on *depth-first search* to generate dialectical trees. As discussed before, such search can be improved by applying $\alpha - \beta$ pruning, so that not every node (argument) is computed. A well-known fact in depth-first search is that the *order* in which branches are generated is important. Fig. 2(b) shows a pruned dialectical tree, where only three arguments were actually computed to deem the root node as defeated. Fig. 2(c) shows that there is an alternative analysis which renders the search space even smaller, by considering first the argument $\langle \mathcal{A}_5, \sim engine_ok, 0.3 \rangle$ instead of $\langle \mathcal{A}_2, \sim fuel_ok, 0.6 \rangle$. Such evaluation order for generating argumentation lines is an issue not taken into account in existing formalizations of argumentation frameworks which mostly rely on dialectical trees computed exhaustively. On the other hand, the actual branching factor of a the dialectical tree is clearly restricted by dialectical constraints as discussed in Def. 6. In order to take into account such features we will introduce some new definitions required to characterize dialectical trees *constructively* rather than *declaratively* as follows.

Definition 10 (Dialectical tree (revisited)). *Consider the definition of dialectical tree (as in Def. 8)* without *the restriction of argumentation lines being exhaustive. A dialectical tree $\mathcal{T}_{\langle \mathcal{A}_0, Q_0, \alpha_0 \rangle}$ will be called* exhaustive *iff each of its argumentation lines is exhaustive, otherwise $\mathcal{T}_{\langle \mathcal{A}_0, Q_0, \alpha_0 \rangle}$ will be called* partial. *We will write $\mathfrak{Tree}_{\mathcal{P}}$ (or just $\mathfrak{Tree}$) to denote the set of all possible dialectical trees based on $\mathcal{P}$.*

In this new setting the process of building a dialectical tree can be thought of as a *computation* starting from an initial tree (consisting of a single node), evolving into more complex trees by adding stepwise new arguments (nodes). This will be formalized by means of a precedence relationship "$\sqsubset$" among trees:

Definition 11 (Precedence relationship $\sqsubset$). *Let $\mathcal{P}$ be a program, and let $\mathcal{T}$, $\mathcal{T}'$ be dialectical trees in $\mathcal{P}$. We define a relationship $\sqsubset \subseteq \mathfrak{Tree} \times \mathfrak{Tree}$, where $\mathcal{T} \sqsubset \mathcal{T}'$*

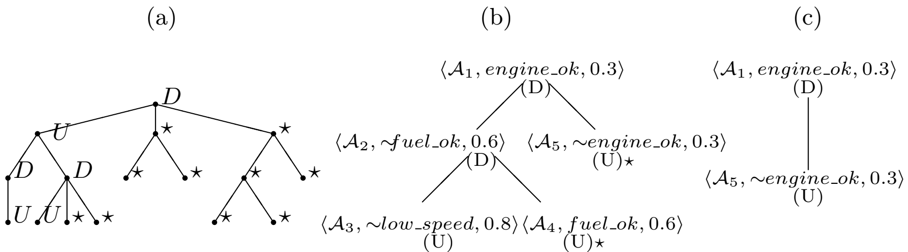


Fig. 2. (a) Dialectical tree, where $\star$'s denote arguments that do not need to be generated because of $\alpha - \beta$ pruning; (b) Dialectical Tree $\mathcal{T}_{\langle \mathcal{A}_1, engine_ok, 0.3 \rangle}$ with exhaustive argumentation lines (ex. 5) marked with $\alpha - \beta$ pruning; (c) Optimally settled dialectical tree $\mathcal{T}'$

(expressed as $\mathcal{T}'$ evolves from *$\mathcal{T}$) whenever $\mathcal{T}'$ can be obtained from $\mathcal{T}$ by extending some argumentation line in $\mathcal{T}$. We will also write $\mathcal{T} \sqsubset_* \mathcal{T}'$ iff there exists a (possibly empty) sequence $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_k$ such that $\mathcal{T} = \mathcal{T}_1 \sqsubset \ldots \sqsubset \mathcal{T}_k = \mathcal{T}'$.*

Clearly from Defs. 7 and 10 the notion of exhaustive dialectical tree can be recast as follows: A dialectical tree $\mathcal{T}_i$ is exhaustive iff there is no $\mathcal{T}_j \neq \mathcal{T}_i$ such that $\mathcal{T}_i \sqsubset_* \mathcal{T}_j$. In fact, every dialectical tree $\mathcal{T}_i$ can be seen as a 'snapshot' of the status of a disputation between two parties (proponent and opponent), and the relationship "$\sqsubset$" allows to capture the evolution of such disputation. As discussed before, pruning strategies could be applied (*e.g.* $\alpha - \beta$ pruning), allowing to determine whether the marking of the root of a partial tree without computing its associated exhaustive tree. We formalize this situation as follows:

Definition 12 (Settled dialectical tree). *Let $\mathcal{T}_i$ be a dialectical tree, such that for every $\mathcal{T}_j$ evolving from $\mathcal{T}_i$ (i.e., $\mathcal{T}_i \sqsubset_* \mathcal{T}_j$) it holds that $Mark(\mathcal{T}_i) = Mark(\mathcal{T}_j)$. Then $\mathcal{T}_i$ is a* settled dialectical tree*. A $\mathcal{T}_i$ is an* optimally settled dialectical tree *iff there is no $\mathcal{T}_i{}' \sqsubset_* \mathcal{T}_i$ such that $\mathcal{T}_i{}'$ is a settled dialectical tree.*

Example 7. Consider the dialectical trees shown in Fig. 2(b) and (c). Then it holds that $\mathcal{T}\,' \sqsubset_* \mathcal{T}_{\langle \mathcal{A}_1, engine_ok, 0.3 \rangle}$. Note also that both $\mathcal{T}_{\langle \mathcal{A}_1, engine_ok, 0.3 \rangle}$ and $\mathcal{T}\,'$ are settled dialectical trees. In particular, $\mathcal{T}\,'$ is an optimally settled dialectical tree.

Note that from the above definition argumentation lines in a settled dialectical tree are not necessarily exhaustive. It is also clear that every exhaustive dialectical tree will be settled, although not necessarily optimally settled. Optimally settled dialectical trees are those involving *the least number of arguments* needed to determine whether the root of the tree is ultimately defeated or not according to the marking procedure.

Proposition 1. *Let $\mathcal{P}$ be a program, and $\langle \mathcal{A}_0, Q_0, \alpha_0 \rangle$ an argument in $\mathcal{P}$. Then $\langle \mathcal{A}_0, Q_0, \alpha_0 \rangle$ is warranted with necessity degree α_0 iff $Mark(\mathcal{T}_{\langle \mathcal{A}_0, Q_0, \alpha_0 \rangle}) = U$, where $\mathcal{T}_{\langle \mathcal{A}_0, Q_0, \alpha_0 \rangle}$ is a settled dialectical tree.*

Next we will analyze how to characterize the computation of dialectical trees in depth-first fashion, modelling informed search oriented towards computing optimally settled dialectical trees. Consider a leaf (argument) $\langle \mathcal{B}, H, \beta \rangle$ in a given argumentation line λ in a partial dialectical tree $\mathcal{T}$ which is not settled, so that further computation will be needed (possibly expanding λ). Clearly, the dialectical constraints given in Def. 6 make that not every defeater as defined in Def. 5 can be used to extend λ. Defeaters satisfying dialectical constraints will be called *feasible defeaters*. Formally:

Definition 13 (Feasible defeaters). *Let $\mathcal{T}_1$ be a partial dialectical tree and let $\langle \mathcal{B}, H, \beta \rangle$ be a leaf node in $\mathcal{T}_1$ at level k in an argumentation line λ. Let $\mathcal{T}$ be the exhaustive dialectical tree associated with $\mathcal{T}_1$ and let $\{\lambda_1, \ldots \lambda_m\}$ be the set of all possible argumentation lines in $\mathcal{T}$ of length $> k+1$ that extend λ, i.e. each λ_i has the form $[\langle \mathcal{A}_0, Q_0, \alpha_0 \rangle, \ldots, \langle \mathcal{B}, H, \beta \rangle, \langle \mathcal{B}_i, H_i, \beta_i \rangle, \ldots]$, for $i = 1 \ldots m$. We define the set of* feasible defeaters *for $\langle \mathcal{B}, H, \beta \rangle$ wrt λ as $\mathsf{FDefeat}(\langle \mathcal{B}, H, \beta \rangle, \lambda) = \bigcup_{i=1}^{m} \{\langle \mathcal{B}_i, H_i, \beta_i \rangle\}$.*

Our depth-first approach can thus be improved by restricting search to feasible defeaters. Note that in depth-first search there will be always one current path associated with the last argument introduced. We call that path *current argumentation line*. Clearly, if $\langle \mathcal{B}, H, \beta \rangle$ is a leaf in the current argumentation line λ associated with the computation of a settled dialectical tree $\mathcal{T}$, *any* element in $\mathsf{FDefeat}(\langle \mathcal{B}, H, \beta \rangle, \lambda)$ is a possible candidate for expanding λ. The marking of the tree $\mathcal{T}$ induces an order "$\prec_{eval}$" in $\mathsf{FDefeat}(\langle \mathcal{B}, H, \beta \rangle, \lambda)$: for any two arguments $\langle \mathcal{B}_i, H_i, \beta_i \rangle$, $\langle \mathcal{B}_j, H_j, \beta_j \rangle$ in $\mathsf{FDefeat}(\langle \mathcal{B}, H, \beta \rangle, \lambda)$, we will say that $\langle \mathcal{B}_i, H_i, \beta_i \rangle \prec_{eval} \langle \mathcal{B}_j, H_j, \beta_j \rangle$ if the subtree rooted in $\langle \mathcal{B}_i, H_i, \beta_i \rangle$ is marked before than the subtree rooted in $\langle \mathcal{B}_j, H_j, \beta_j \rangle$. Fig. 3 illustrates how a dialectical tree can be built in a depth-first fashion using $\alpha - \beta$ pruning and the evaluation order $\prec_{eval}$. In order to speed up the construction of a settled dialectical tree, our approach will be twofold: on the one hand, we will identify which literals can be deemed as candidates for computing feasible defeaters. On the other hand, we will provide a definition of $\prec_{eval}$ which prunes the search space using dialectical constraints.

Algorithm **1** BuildDialecticalTree
Input: $\langle \mathcal{A}, Q, \alpha \rangle$, $\lambda = [\langle \mathcal{A}, Q, \alpha \rangle]$ Output: $\mathcal{T}_{\langle \mathcal{A}, Q, \alpha \rangle}$, *Mark (Marking)*
{*uses* α-β *pruning and evaluation ordering* $\preceq_{eval}$}
Global variable: $\mathcal{T}_{\langle \mathcal{A}, Q, \alpha \rangle}$ Local variables: $MarkAux$, ParentDefeated, λ
{λ *is the current argumentation line, initially* $\lambda = [\langle \mathcal{A}, Q, \alpha \rangle]$}

Put $\langle \mathcal{A}, Q, \alpha \rangle$ as root node of $\mathcal{T}_{\langle \mathcal{A}, Q, \alpha \rangle}$
Compute $\mathsf{FDefeat}(\langle \mathcal{A}, Q, \alpha \rangle, \lambda) = \{\langle \mathcal{A}_0, Q_0, \alpha_0 \rangle, \ldots, \langle \mathcal{A}_k, Q_k, \alpha_0 \rangle\}$
{$\mathsf{FDefeat}(\langle \mathcal{A}, Q, \alpha \rangle, \lambda)$= *feasible defeaters for* $\langle \mathcal{A}, Q, \alpha \rangle$ *wrt* λ}
If $S \neq \emptyset$
  Then
    ParentDefeated := false
    While (ParentDefeated=false) **and** ($S \neq \emptyset$) **do**
      Choose some $\langle \mathcal{A}_i, Q_i, \alpha_i \rangle \in S$ minimal wrt $\prec_{eval}$
      $S := S \setminus \{\langle \mathcal{A}_i, Q_i, \alpha_i \rangle\}$
      $\lambda := \lambda \circ \langle \mathcal{A}_i, Q_i, \alpha_i \rangle$ {*expand* λ *adding new argument* $\langle \mathcal{A}_i, Q_i, \alpha_i \rangle$}
      BuildDialecticalTree($\langle \mathcal{A}_i, Q_i, \alpha_i \rangle, \mathcal{T}_{\langle \mathcal{A}_i, Q_i, \alpha_i \rangle}, MarkAux$)
      Add $\mathcal{T}_{\langle \mathcal{A}_i, Q_i, \alpha_i \rangle}$ as immediate subtree of $\langle \mathcal{A}, Q, \alpha \rangle$.
      If $MarkAux$=U **then** ParentDefeated := true
    end **while**
    If ParentDefeated=false {$S = \emptyset$, *all defeaters were defeated*}
     then $Mark$:= U {*mark* $\mathcal{T}_{\langle \mathcal{A}, Q, \alpha \rangle}$ *as* U}
     else $Mark$:= D {*mark* $\mathcal{T}_{\langle \mathcal{A}, Q, \alpha \rangle}$ *as* D}
  else {$S = \emptyset$, *hence* $\langle \mathcal{A}, Q, \alpha \rangle$ *has no defeaters*}
    $Mark$:= U {*mark* $\mathcal{T}_{\langle \mathcal{A}, Q, \alpha \rangle}$ *as* U}
Return $\mathcal{T}_{\langle \mathcal{A}, Q, \alpha \rangle}, Mark$

Fig. 3. Algorithm for building and labelling settled dialectical trees in a depth-first fashion taking into account feasible defeaters and evaluation order $\prec_{eval}$

4 Pruning Dialectical Trees in P-DeLP

Given an argument $\langle \mathcal{A}_0, Q_0, \alpha_0 \rangle$, building a dialectical tree $\mathcal{T}_{\langle \mathcal{A}_0, Q_0, \alpha_0 \rangle}$ involves computing defeaters in a recursive way. According to Def. 4, to automate the computation of such defeaters it is necessary to detect the set of *disagreement literals* $\{ (L_1, \phi_1), \ldots, (L_k, \phi_k) \}$ that can be source of conflict with counter-arguments $\langle \mathcal{B}_1, H_1, \beta_1 \rangle, \ldots, \langle \mathcal{B}_k, H_k, \beta_k \rangle$ that defeat $\langle \mathcal{A}_0, Q_0, \alpha_0 \rangle$. Fortunately, in the context of P-DeLP this can be done on the basis of the *consequents* of uncertain clauses associated with subarguments in $\langle \mathcal{A}_0, Q_0, \alpha_0 \rangle$.

Definition 14 (Set of consequents Co). *Let $\langle \mathcal{A}_0, Q_0, \alpha_0 \rangle$ be an argument. The set $\mathbf{Co}(\langle \mathcal{A}_0, Q_0, \alpha_0 \rangle) = \{ (Q, \alpha) \mid \exists\ (Q \leftarrow L_1 \wedge L_2 \wedge \ldots \wedge L_k, \gamma) \in \mathcal{A}_0$ such that $\langle \mathcal{A}, Q, \alpha \rangle$ is a subargument of $\langle \mathcal{A}_0, Q_0, \alpha_0 \rangle \}$. We generalize this to a set S of arguments, $S = \bigcup_{i=1\ldots k} \langle \mathcal{A}_i, Q_i, \alpha_i \rangle$, defining $\mathbf{Co}(S) = \bigcup_{i=1\ldots k} \mathbf{Co}(\langle \mathcal{A}_i, Q_i, \alpha_i \rangle)$.*[8]

Lemma 1 (Goal-driven defeat [9]). *Let $\langle \mathcal{A}, Q, \alpha \rangle$ be an argument, and let $\langle \mathcal{B}, H, \beta \rangle$ be a defeater for $\langle \mathcal{A}, Q, \alpha \rangle$. Then there exists an argument $\langle \mathcal{B}, H', \beta \rangle$, such that H' is the complement of a literal in $\mathbf{Co}(\langle \mathcal{A}, Q, \alpha \rangle)$, (where complement of (L, γ) is defined as $(\sim L, \gamma)$).*

Lemma 1 allows to search for defeaters automatically by backward chaining, on the basis of the consequents of uncertain clauses in an argument. Thus if $(L, \gamma) \in \mathbf{Co}(\langle \mathcal{A}_0, Q_0, \alpha_0 \rangle)$, a search for a defeater with disagreement literal (L, ϕ) will involve finding an argument for concluding $(\sim L, \gamma')$, with $\gamma' \geq \gamma$.

Example 8. Consider $\langle \mathcal{A}_1, engine_ok, 0.3 \rangle$ in Ex. 2. Then $\mathbf{Co}(\langle \mathcal{A}_1, engine_ok, 0.3 \rangle) = \{ (engine_ok, 0.3), (pump_fuel, 0.6), (fuel_ok, 0.3), (oil_ok, 0.3), (pump_oil, 0.8) \}$. Defeaters for $\langle \mathcal{A}_1, engine_ok, 0.3 \rangle$ can be found by backward chaining from the complement of each weighted literal (L, γ), searching for arguments for $(\sim L, \gamma')$, with $\gamma' \geq \gamma$.

From the above considerations we can establish the following inclusionship for detecting candidate disagreement literals in an argument $\langle \mathcal{A}, Q, \alpha \rangle$ appearing as a leaf in an argumentation line λ:

$$\mathbf{Optimal}(\langle \mathcal{A}, Q, \alpha \rangle, \lambda) \subseteq \mathbf{Feasible}(\langle \mathcal{A}, Q, \alpha \rangle, \lambda) \subseteq \mathbf{Co}(\langle \mathcal{A}, Q, \alpha \rangle)$$

Here $\mathbf{Feasible}(\langle \mathcal{A}, Q, \alpha \rangle, \lambda)$ denotes the set of weighted literals (ϕ, α) which are possible disagreement literals for $\langle \mathcal{A}, Q, \alpha \rangle$ for some *feasible* defeater, whereas $\mathbf{Optimal}(\langle \mathcal{A}, Q, \alpha \rangle, \lambda)$ denotes the set of weighted literals (ϕ, α) which are possible disagreement literals for feasible defeaters leading to *the shortest argumentation lines.*[9] Clearly $\mathbf{Optimal}(\langle \mathcal{A}, Q, \alpha \rangle, \lambda)$ is in a sense an *ideal* set of disagreement literals, for which we can find different approximations. One possibility is to consider the set $\mathbf{Feasible}(\langle \mathcal{A}, Q, \alpha \rangle, \lambda)$. However, determining feasible defeaters is computationally also quite a difficult task, as it involves checking different

[8] Note that the set $\mathbf{Co}(\langle \mathcal{A}_0, Q_0, \alpha_0 \rangle)$ can be easily computed along the derivation process of the argument itself.

[9] Note that this is a set, as there may be different argumentation lines extending λ, all of them having the same length.

dialectical constraints (see Def. 6). As discussed before, a more tractable way to detect candidate defeaters is to consider the set $\mathbf{Co}(\langle \mathcal{A}, Q, \alpha \rangle)$.

A better approximation than $\mathbf{Co}(\langle \mathcal{A}, Q, \alpha \rangle)$ can be stated taking into account the following intuition: let us assume that the current argumentation line λ has been computed up to level k. From the 'non-contradiction' constraint, even-level as well as odd-level arguments in λ should not be contradictory. This accounts to saying also that literals which are common to *both* even-level and odd-level arguments cannot be disagreement literals within any extension of λ. To formalize this notion, we will suitably extend the definitions for set intersection and difference for weighted literals as follows: given two sets of weighted literals S_1 and S_2, we define intersection among S_1 and S_2 as $S_1 \sqcap S_2 =_{def} \{(Q, \alpha) \mid (Q, \alpha_1) \in S_1$ and $(Q, \alpha_2) \in S_2$, with $\alpha = min(\alpha_1, \alpha_2)$ $\}$. Similarly, we define difference among S_1 and S_2 as follows: $S_1 \setminus S_2 =_{def} \{ (Q, \alpha) \mid (Q, \alpha) \in S_1$ and $\nexists (Q, \beta) \in S_2$, for $\beta > 0$ $\}$

Definition 15 (Set $\mathsf{SharedLit}$). *Let λ be an argumentation line.* $\mathsf{SharedLit}(\lambda, k)$ *is the set of weighted literals common to even-level and odd-level arguments up to level k, i.e.* $\mathsf{SharedLit}(\lambda, k) =_{def} \mathbf{Co}(S^k_\lambda) \sqcap \mathbf{Co}(I^k_\lambda)$.

Proposition 2. *Let λ be an argumentation line in a partial dialectical tree $\mathcal{T}$, and let $\langle \mathcal{A}, Q, \alpha \rangle$ an argument which is a leaf in λ at level k. Let $(L, \gamma) \in \mathsf{SharedLit}(\lambda, k), k > 0$. Then $(L, \gamma') \notin \mathbf{Feasible}(\langle \mathcal{A}, Q, \alpha \rangle, \lambda)$, for any $\gamma' \geq \gamma$.*

Proposition 2 allows to further refine the inclusion relationship given before as follows:

$$\mathbf{Feasible}(\langle \mathcal{A}, Q, \alpha \rangle, \lambda) \subseteq \mathbf{Co}(\langle \mathcal{A}, Q, \alpha \rangle) \setminus \mathsf{SharedLit}(\lambda, k) \subseteq \mathbf{Co}(\langle \mathcal{A}, Q, \alpha \rangle)$$

We can now come back to the original question: how to choose which defeater belongs to the (on the average) shorter argumentation line, *i.e.* the one more prone to settle the disputation as soon as possible. From our preceding results we can suggest the following definition for $\prec_{eval}$:

Definition 16 (Evaluation order based on $\mathsf{SharedLit}$). *Let λ be an argumentation line, and let $\langle \mathcal{A}, Q, \alpha \rangle$ be a leaf at level k. Let $\langle \mathcal{A}_1, Q_1, \alpha_1 \rangle$ and $\langle \mathcal{A}_2, Q_2, \alpha_2 \rangle$ be two candidate defeaters for $\langle \mathcal{A}, Q, \alpha \rangle$, such that λ can be extended to λ_1 (using $\langle \mathcal{A}_1, Q_1, \alpha_1 \rangle$) or λ_2 (using $\langle \mathcal{A}_2, Q_2, \alpha_2 \rangle$). Then $\langle \mathcal{A}_1, Q_1, \alpha_1 \rangle \prec_{eval} \langle \mathcal{A}_2, Q_2, \alpha_2 \rangle$ iff* $\mathbf{Co}(\langle \mathcal{A}_1, Q_1, \alpha_1 \rangle) \setminus \mathsf{SharedLit}(\lambda_1, k+1) \subseteq \mathbf{Co}(\langle \mathcal{A}_2, Q_2, \alpha_2 \rangle) \setminus \mathsf{SharedLit}(\lambda_2, k+1)$.[10]

Example 9. Consider the argument $\langle \mathcal{A}_1, engine_ok, 0.3 \rangle$ as in Ex. 2 and 8, and assume that the current argumentation line is $\lambda = [\langle \mathcal{A}_1, engine_ok, 0.3 \rangle]$. In such a case the set $\mathsf{FDefeat}(\langle \mathcal{A}_1, engine_ok, 0.3 \rangle, \lambda)$ has two defeaters $\{ \langle \mathcal{A}_2, \sim fuel_ok, 0.6 \rangle, \langle \mathcal{A}_5, \sim engine_ok, 0.3 \rangle \}$, computed in Ex. 3 and 5. Argumentation line λ can therefore be extended in two different ways, $\lambda_1 = \lambda \circ \langle \mathcal{A}_2, \sim fuel_ok, 0.6 \rangle$ and $\lambda_2 = \lambda \circ$

[10] Note that this partial order can be refined by considering those arguments with higher necessity, *i.e.* given two defeaters $\langle \mathcal{A}_1, Q_1, \alpha_1 \rangle$ and $\langle \mathcal{A}_2, Q_2, \alpha_2 \rangle$ equally preferred wrt $\prec_{eval}$, the one having $max(\beta_i, \beta_j)$ as necessity degree is preferred.

$\langle \mathcal{A}_5, \sim engine_ok, 0.3 \rangle$. Let us compute the set of consequents for these arguments:

$$\mathbf{Co}(\langle \mathcal{A}_1, engine_ok, 0.3 \rangle) = \{ (engine_ok, 0.3), (pump_fuel, 0.6), (fuel_ok, 0.3), (oil_ok, 0.3), (pump_oil, 0.8) \}.$$

$$\mathbf{Co}(\langle \mathcal{A}_5, \sim engine_ok, 0.3 \rangle) = \{ (\sim engine_ok, 0.3), (pump_fuel, 0.6), (fuel_ok, 0.3), (oil_ok, 0.3), (pump_oil, 0.8) \}.$$

$$\mathbf{Co}(\langle \mathcal{A}_2, \sim fuel_ok, 0.6 \rangle) = \{ (\sim fuel_ok, 0.6), (low_speed, 0.8), (pump_clog, 0.7) \}$$

From the above sets we have then $\mathsf{SharedLit}(\lambda_1, 1) = \emptyset$ and $\mathsf{SharedLit}(\lambda_2, 1) = \{ (engine_ok, 0.3), (pump_fuel, 0.6), (fuel_ok, 0.3), (oil_ok, 0.3), (pump_oil, 0.8) \}$. Consequently, it holds that

$$\mathbf{Co}(\langle \mathcal{A}_1, engine_ok, 0.3 \rangle) \backslash \mathsf{SharedLit}(\lambda_2, 1) \subset \mathbf{Co}(\langle \mathcal{A}_1, engine_ok, 0.3 \rangle) \backslash \mathsf{SharedLit}(\lambda_1, 1)$$

Thus the defeater $\langle \mathcal{A}_5, \sim engine_ok, 0.3 \rangle$ should be evaluated before the defeater $\langle \mathcal{A}_2, \sim fuel_ok, 0.6 \rangle$ in the depth-first computation of the dialectical tree using the algorithm in Fig. 3.

Although Example 9 is rather naïve, it is intended to show one possible way of characterizing the evaluation order $\prec_{eval}$, reducing the average branching factor of the dialectical tree when in a depth-first fashion.

5 Related Work and Conclusions

In this paper we have presented a novel approach to characterize dialectical reasoning in the context of Possibilistic Defeasible Logic Programming, aiming at speeding up the underlying inference procedure. The contribution of this paper is twofold: on the one hand, we have formalized the notion of dialectical trees constructively, taking into account salient features in modelling the depth-first construction of such trees. On the other hand, we have analyzed the role of dialectical constraints as an additional element for pruning the resulting search space. Although our characterization is based in P-DeLP, it can be generalized to be applied in other argumentation frameworks based on logic programming. It must be remarked that P-DeLP is an extension of Defeasible Logic Programming [9], which has been successfully integrated in a number of real-world applications (e.g. clustering [10], and recommender systems [11]).

Our work complements previous research concerning the dynamics of argumentation, notably [12] and [13]. In particular, Prakken [12] has analyzed the exchange of arguments in the context of dynamic disputes. Our approach can also be understood in the light of his characterization of dialectical proof theories. However, Prakken focuses on a comprehensive but rather general framework, in which important computational issues (*e.g.* detecting disagreement literals, search space considerations, etc.) are not taken into account. Hunter [14] analyzes the search space associated with dialectical trees taking into account novel features such as the *resonance* of arguments. His interesting formalization combines a number of features that allow to assess the impact of dialectical trees, contrasting shallow vs. deep trees. However, computational aspects as the ones analyzed in this paper are outside the scope of his work. In [4] a throughout

analysis of various argumentation semantics for logic programming is presented on the basis of parametric variations of derivation trees. In contrast with that approach, our aim in this paper was not to characterize different emerging semantics, but rather to focus on an efficient construction of dialectical trees for speeding up inference. On the other hand, in [4] the authors concentrate in normal logic programming, whereas our approach deals with extended logic programming enriched with necessity degrees. Recently semantical aspects of P-DeLP have been analyzed in the context of specialized inference operators [15].

It must be remarked that our approach can also be improved by considering the non-circularity constraint (see Def. 6) for argumentation lines: as the current argumentation line λ is computed, the set of feasible defeaters associated to the last argument in λ at level k is also restricted by arguments which already appeared earlier at any level $k' < k$. Part of our current research work involves how to extend our algorithm to include such non-circularity constraints in our analysis, in order to develop a full-fledged implementation of the algorithm presented in this paper including such features. Our experiments so far have been performed only on a "proof of concept" prototype, as we have not been able yet to carry out thorough evaluations in the context of a real-world application. The results obtained, however, have been satisfactory and as stated before can be generalized to most argumentation frameworks. The development of such generalization is part of our future work.

Acknowledgements. This research was partially supported by Projects TIC2003-00950, TIN2004-07933-C03-01/03, by Ramón y Cajal Program (Ministerio de Ciencia y Tecnología, Spain), by CONICET (Argentina), by the Secretaría General de Ciencia y Tecnología de la Universidad Nacional del Sur and by Agencia Nacional de Promoción Científica y Tecnológica (PICT 2002 No. 13096).

References

1. Chesñevar, C.I., Simari, G., Alsinet, T., Godo, L.: A Logic Programming Framework for Possibilistic Argumentation with Vague Knowledge. In: Proc. of the Intl. Conf. in Uncertainty in Art. Intelligence. (UAI 2004). Banff, Canada. (2004) 76–84
2. Alsinet, T., Godo, L.: A complete calculus for possibilistic logic programming with fuzzy propositional variables. In: Proc. of the UAI-2000 Conference. (2000) 1–10
3. Dubois, D., Lang, J., Prade, H.: Possibilistic logic. In D.Gabbay, C.Hogger, J.Robinson, eds.: Handbook of Logic in Art. Int. and Logic Prog. (Nonmonotonic Reasoning and Uncertain Reasoning). Oxford Univ. Press (1994) 439–513
4. Kakas, A., Toni, F.: Computing argumentation in logic programming. Journal of Logic Programming **9** (1999) 515:562
5. Chesñevar, C.I., Maguitman, A., Loui, R.: Logical Models of Argument. ACM Computing Surveys **32** (2000) 337–383
6. Prakken, H., Vreeswijk, G.: Logical Systems for Defeasible Argumentation. In Gabbay, D., F.Guenther, eds.: Handbook of Philosophical Logic. Kluwer Academic Publishers (2002) 219–318

7. Besnard, P., Hunter, A.: A logic-based theory of deductive arguments. Artificial Intelligence **1:2** (2001) 203–235
8. Prakken, H., Sartor, G.: Argument-based extended logic programming with defeasible priorities. Journal of Applied Non-classical Logics **7** (1997) 25–75
9. García, A., Simari, G.: Defeasible Logic Programming: An Argumentative Approach. Theory and Practice of Logic Programming **4** (2004) 95–138
10. Gómez, S., Chesñevar, C.: A Hybrid Approach to Pattern Classification Using Neural Networks and Defeasible Argumentation. In: Proc. of 17th Intl. FLAIRS Conf. Miami, Florida, USA, AAAI Press (2004) 393–398
11. Chesñevar, C., Maguitman, A.: An Argumentative Approach to Assessing Natural Language Usage based on the Web Corpus. In: Proc. of the ECAI-2004 Conference. Valencia, Spain. (2004) 581–585
12. Prakken, H.: Relating protocols for dynamic dispute with logics for defeasible argumentation. Synthese (special issue on New Perspectives in Dialogical Logic) **127** (2001) 187:219
13. Brewka, G.: Dynamic argument systems: A formal model of argumentation processes based on situation calculus. J. of Logic and Computation **11** (2001) 257–282
14. Hunter, A.: Towards Higher Impact Argumentation. In: Proc. of the 19th American National Conf. on Artificial Intelligence (AAAI'2004), MIT Press (2004) 275–280
15. Chesñevar, C., Simari, G., Godo, L., Alsinet, T.: Argument-based expansion operators in possibilistic defeasible logic programming: Characterization and logical properties. In: 8th European Conf. on Symbolic and Qualitative Aspects of Reasoning Under Uncertainty (ECSQARU 2005), Barcelona, Spain (to appear). (2005)

An Approximation of Action Theories of $\mathcal{AL}$ and Its Application to Conformant Planning

Tran Cao Son[1], Phan Huy Tu[1], Michael Gelfond[2], and A. Ricardo Morales[2]

[1] Department of Computer Science, New Mexico State University,
Las Cruces, NM 88003, USA
{tson, tphan}@cs.nmsu.edu
[2] Department of Computer Science, Texas Tech University, Lubbock, TX 79409, USA
{mgelfond, ricardo}@cs.ttu.edu

Abstract. In this paper we generalize the notion of approximation of action theories introduced in [13,26]. We introduce a logic programming based method for constructing approximation of action theories of $\mathcal{AL}$ and prove its soundness. We describe an approximation based conformant planner and compare its performance with other state-of-the-art conformant planners.

1 Introduction and Motivation

Static causal laws (a.k.a. *state constraints* or *axioms*) constitute an important part of every dynamic domain. Unlike an effect of an action, a static causal law represents a relationship between fluents. For example,

(a) In the travel domain, the static causal law "*one person cannot be at A if he is at B*" states that $at(B)$ is false if $at(A)$ is true;
(b) In the block world domain, the static causal law "*block A is above block B if A is on B*" says that $above(A, B)$ is true if $on(A, B)$ is true;

Static causal laws can cause actions to have *indirect effects*. For example, the action of putting the block A atop the block B, denoted by $put(A, B)$, causes $on(A, B)$ to be true. The static causal law (b) implies that $above(A, B)$ is also true, i.e., $above(A, B)$ is an indirect effect of $put(A, B)$. The problem of determining such indirect effects is known as the *ramification problem* in the area of reasoning about action and change (RAC).

In the last decade, several solutions to the ramification problem have been proposed. Each of these solutions extends a framework for RAC to allow static causal laws [2,18,22,23,20,15,17]. While being intensively studied by the RAC's research community, static causal laws have rarely been *directly* considered by the planning community. Although the original specification of the Planning Domain Description Language (PDDL) – a language frequently used for the specification of planning problems by the planning community – includes axioms (or static causal laws in our notation) [14], most of the planning domains used in the recent planning competitions [1,19,11] do not include axioms. The main reason for this practice is that it is widely believed that axioms can be compiled into actions' effect propositions; thus, making the representation of and reasoning about axioms become unnecessary in planning. This is partly true due to the fact that PDDL only allows non-recursive axioms. In a recent paper [29], it is proved

C. Baral et al. (Eds.): LPNMR 2005, LNAI 3662, pp. 172–184, 2005.
© Springer-Verlag Berlin Heidelberg 2005

that adding axioms to the planning language not only improves the readability and elegance of the representation but also increases the expressiveness of the language. It is also shown that the addition of a component to handle axioms in a planner can indeed improve the performance of the planner.

The main difficulty in planning in domains with static causal laws lies directly in defining and computing the successor states. In general, domains with static causal laws are nondeterministic; for example, in a theory with a single action a and three fluents f, g, and h with the property that execution of a causes f to become true and the two static causal laws

(i) if f is true and g is false then h must be true; and,
(ii) if f is true and h is false then g must be true.

Intuitively, the execution of a in a state where f, h, and g are false will yield two possible states. In one state, f and g are true and h is false. In the other one, f and h are true and g is false. This nondeterminism leads to the fact that the execution of an action sequence can generate different trajectories. Thus, exact planning[1] is similar to conformant planning, an approach to dealing with incomplete information in planning. It is also worth noticing that the complexity of conformant planning (Σ_2^P) is much higher than planning in deterministic domains (**NP**-complete) [3,30]. It is also pointed out in [3] that approximations of the transition function between states can help reduce the complexity of the planning problem.

In this paper, we further investigate the notion of approximations of action theories introduced in [26,13]. We define an approximation for action theories of $\mathcal{AL}$. The key difference between the newly developed approximation and those proposed in [26,13] is that it is applicable for action descriptions with arbitrary static causal laws: while the approximation proposed in [13] is only for specific type of state constraints, the approximations in [26] are defined for action descriptions with sensing actions but without state constraints. We use a logic program in defining the approximation.

The paper is organized as follows. In the next section, we review the basics of the language $\mathcal{AL}$. Afterward, we define an approximation of $\mathcal{AL}$ action theories. We then proceed with the description of a logic programming based conformant planner which makes use of the approximation. We then compare the performance of our planner with some conformant planners which are closely related to our planner.

2 Syntax and Semantics of $\mathcal{AL}$

We consider domains which can be represented by a transition diagram whose nodes are possible states of the domain and whose arcs are actions that take the domain from one state to another. Paths of the diagram correspond to possible trajectories of the system. We limit our attention to transition diagrams which can be defined by action descriptions of the action language $\mathcal{AL}$ from [4]. The signature Σ of an action description of $\mathcal{AL}$ consists of two disjoint, non-empty sets of symbols: the set $\mathbf{F}$ of fluents, and

[1] By exact planning we mean the problem of finding a polynomial-bounded length sequence of actions that can achieve the goal at the end of every possible trajectory generated by the action sequence.

the set **A** of *elementary actions*. By an *action* we mean a non-empty set a of elementary actions. Informally we interpret an execution of a as a simultaneous execution of its components. For simplicity we identify an elementary action e with $\{e\}$. By *fluent literals* we mean fluents and their negations. By $\bar{l}$ we denote the fluent literal complementary to l. A set S of fluent literals is called *complete* if, for any $f \in \mathbf{F}$, $f \in S$ or $\neg f \in S$. An action description $\mathcal{D}$ of $\mathcal{AL}$ is a collection of statements of the form:

$$e \textbf{ causes } l \textbf{ if } p \tag{1}$$

$$l \textbf{ if } p \tag{2}$$

$$\textbf{impossible } a \textbf{ if } p \tag{3}$$

where e is an elementary action, a is an action, l is a fluent literal, and p is a set of fluent literals from the signature Σ of $\mathcal{D}$. The set p is often referred to as the *precondition* of the corresponding statement. When it is empty, the "if" part of the statement can be omitted. Statement (1), called a *dynamic causal law*, says that, if e is executed in a state satisfying p then l will hold in any resulting state. Statement (2), called a *static causal law*, says that any state satisfying p must satisfy l. Statement (3) is an *impossibility condition*. It says that action a cannot be performed in a state satisfying p. We next define the transition diagram, $T(\mathcal{D})$ specified by an action description $\mathcal{D}$ of $\mathcal{AL}$.

A set of literals S is *closed* under a static causal law (2) if $l \in S$ whenever $p \subseteq S$. By $Cn(S)$, we denote the smallest set of literals that contains S and is closed under the static causal laws of $\mathcal{D}$. A *state* σ of $T(\mathcal{D})$ is a complete, consistent set of literals closed under the static causal laws of $\mathcal{D}$. An action b is said to be *prohibited* in σ if $\mathcal{D}$ contains an impossibility condition (3) such that $p \subseteq \sigma$ and $a \subseteq b$. $E(a, \sigma)$ stands for the set of all fluent literals l for which there is a causal law (1) in $\mathcal{D}$ such that $p \subseteq \sigma$ and $e \in a$. Elements of $E(a, \sigma)$ are called *direct effects* of the execution of a in σ.

Definition 1 ([21]). For an action a and two states σ_1 and σ_2, a transition $\langle \sigma_1, a, \sigma_2 \rangle \in T(\mathcal{D})$ iff a is not prohibited in σ_1 and $\sigma_2 = Cn(E(a, \sigma_1) \cup (\sigma_1 \cap \sigma_2))$.

An alternate sequence of states and actions, $M = \langle \sigma_0, a_0, \sigma_1, \ldots, a_{n-1}, \sigma_n \rangle$, is a *path* in a transition diagram $T(\mathcal{D})$ if $\langle \sigma_i, a_i, \sigma_{i+1} \rangle \in T(\mathcal{D})$ for $0 \leq i < n$. M is called a *model* of the chain of events $\alpha = \langle a_0, \ldots, a_{n-1} \rangle$; σ_0 (resp. σ_n) is referred to as the *initial state* (resp. *final state*) of M; M *entails* a set of fluent literals s, written as $M \models s$, if $s \subseteq \sigma_n$. We sometime write $\langle \sigma_0, \alpha, \sigma_n \rangle \in T(\mathcal{D})$ to denote that there exists a model of α whose initial state and final state is σ_0 and σ_n, respectively. An action description $\mathcal{D}$ is called *deterministic* if for any state σ_1 and action a there is at most one successor state σ_2 such that $\langle \sigma_1, a, \sigma_2 \rangle \in T(\mathcal{D})$. Note that if $\mathcal{D}$ is deterministic there can be at most one model for α given the initial state σ_0 and final state σ_n. We denote this model by $\sigma_n = \alpha(\sigma_0)$. Notice that in the presence of static causal laws, action theories can be nondeterministic. As an example, the second theory in the introduction can be described by the action description $\mathcal{D}_0$ consisting of the following statements:

$$\mathcal{D}_0 = \left\{ a \textbf{ causes } f \qquad g \textbf{ if } f, \neg h \qquad h \textbf{ if } f, \neg g \right\}$$

Observe that $T(\mathcal{D}_0)$ includes the transitions $\langle \{\neg f, \neg h, \neg g\}, a, \{f, h, \neg g\} \rangle$ and $\langle \{\neg f, \neg h, \neg g\}, a, \{f, g, \neg h\} \rangle$. Hence, $\mathcal{D}_0$ is non-deterministic.

An action a is *executable* in state σ_1 if there is a state σ_2 such that $\langle \sigma_1, a, \sigma_2 \rangle \in T(\mathcal{D})$; a chain of events $\alpha = \langle a_1, \ldots, a_{n-1} \rangle$ is executable in a state σ if there exists a

path $\langle \sigma, \alpha, \sigma' \rangle$ in $T(\mathcal{D})$ for some σ'; $\mathcal{D}$ is called *consistent* if for any state σ_1 and action a which is not prohibited in σ_1 there exists at least one successor state σ_2 such that $\langle \sigma_1, a, \sigma_2 \rangle \in T(\mathcal{D})$.

3 Approximating Action Theories of $\mathcal{AL}$

Normally an agent does not have complete information about its current state. Instead its knowledge is limited to the current *partial state* – a consistent collection of fluent literals closed under the static causal laws of the agent's action description $\mathcal{D}$. In what follows partial states and states are denoted by (possibly indexed) letters s and σ respectively.

A state σ that includes a partial state s is called a *completion* of s. By $comp(s)$ we denote the set of all completions of s. An action a is *safe* in s if it is executable in every completion of s. A chain of events $\alpha = \langle a_0, \ldots, a_{n-1} \rangle$ is *safe* in s if (*i*) a_0 is safe in s; and (*ii*) for every state σ' such that $\langle \sigma, a_0, \sigma' \rangle \in T(\mathcal{D})$ for some $\sigma \in comp(s)$, $\langle a_1, \ldots, a_{n-1} \rangle$ is safe in σ'.

For many of its reasoning tasks the agent may need to know the effects of its actions which are determined by the fluents from s (as opposed to the actual completion of s). In [26] the authors suggest to model such knowledge by a transition function which approximates the transition diagram $T(\mathcal{D})$ for deterministic action theories with sensing actions. We will next generalize this notion to action theories in $\mathcal{AL}$. Even though approximations can be non-deterministic, in this paper we will be interested only in deterministic approximations.

Definition 2 (Approximation). $T'(\mathcal{D})$ is an *approximation* of $T(\mathcal{D})$ if

1. States of $T'(\mathcal{D})$ are partial states of $T(\mathcal{D})$.
2. If $\langle s, a, s' \rangle \in T'(\mathcal{D})$ then for every $\sigma \in comp(s)$,
   (a) a is executable in σ and,
   (b) $s' \subseteq \sigma'$ for every σ' such that $\langle \sigma, a, \sigma' \rangle \in T(\mathcal{D})$.

An approximation $T'(\mathcal{D})$ is *deterministic* if for each partial state s and action a, there exists at most one s' such that $\langle s, a, s' \rangle \in T'(\mathcal{D})$. The next observation shows that an approximation must be sound.

Observation 1. *Let $T'(\mathcal{D})$ be an approximation of $T(\mathcal{D})$. Then, for every chain of events α if $\langle s, \alpha, s' \rangle \in T'(\mathcal{D})$ then for every $\sigma \in comp(s)$, (a) α is executable in σ; and (b) $s' \subseteq \sigma'$ for every σ' such that $\langle \sigma, \alpha, \sigma' \rangle \in T(\mathcal{D})$.*

In what follows we describe a method for constructing approximations of action theories of $\mathcal{AL}$. In our approach, the transitions in $T'(\mathcal{D})$ will be defined by a logic program $\pi(\mathcal{D})$ called the *cautious encoding* of $\mathcal{D}$. The signature of $\pi(\mathcal{D})$ includes terms corresponding to fluent literals and actions of $\mathcal{D}$, as well as non-negative integers used to represent time steps. For convenience, we often write $\pi(\mathcal{D}, n)$ to denote the program $\pi(\mathcal{D})$ where the time constants take values between 0 and n. Atoms of $\pi(\mathcal{D})$ are formed by the following (sorted) predicate symbols:

– $h(l, T)$ is true if literal l holds at time-step T;
– $o(e, T)$ is true if action e occurs at time-step T;

- $dc(l, T)$ is true if literal l is a direct effect of an action that occurs at time $T{-}1$; and
- $ph(l, T)$ is true if literal l possibly holds at time T.

The program also contains a set of auxiliary predicates, including $time$, $fluent$, and $action$, for enumerating constants of sorts time, fluent, and action respectively; $literal$ and $contrary$ for defining literals and complementary literals, respectively[2].

In our representation, letters T, F, L, and A (possibly indexed) are used to represent variables of sorts time, fluent, literal, and action correspondingly. Morever, we also use some shorthands: if a is an action then $o(a, T) = \{o(e, T) : e \in a\}$. For a set of fluent literals p, and a predicate symbol $\rho \in \{h, dc, ph\}$, $\rho(p, T) = \{\rho(l, T) : l \in p\}$ and $not\ \rho(p, T) = \{not\ \rho(l, T) : l \in p\}$. For a fluent f, by $\overline{l}$ we mean $\neg f$ if $l = f$ and f if $l = \neg f$. Literals $\overline{l}$ and l are called contrary literals. For a set of literals p, $\overline{p} = \{\overline{l} : l \in p\}$. The set of rules of $\pi(\mathcal{D})$ consists of those encoding the laws in $\mathcal{D}$, those encoding the inertial axioms, and some auxiliary rules. We next describe these subsets of rules:

1. For each dynamic causal law (1) in $\mathcal{D}$, the rules

$$h(l, T{+}1) \leftarrow o(e, T), h(p, T) \tag{4}$$

$$dc(l, T + 1) \leftarrow o(e, T), h(p, T) \tag{5}$$

belong to $\pi(\mathcal{D})$. The first rule states that l holds at $T + 1$ if e occurs at T and the condition p holds at T. The second rule indicates that l is a direct effect of the execution of e. Since the state at the time moment T might be incomplete, we add to $\pi(\mathcal{D})$ the rule

$$ph(l, T + 1) \leftarrow o(e, T), not\ h(\overline{p}, T) \tag{6}$$

which says that l might hold at $T{+}1$ if e occurs at T and the precondition p possibly holds at T.

2. For each static causal law (2) in $\mathcal{D}$, $\pi(\mathcal{D})$ contains the two rules:

$$h(l, T) \leftarrow h(p, T) \tag{7}$$

$$ph(l, T) \leftarrow ph(p, T) \tag{8}$$

These rules basically state that if p holds (or possibly holds) at T then so does l.

3. For each impossibility condition (3) in $\mathcal{D}$, we add to $\pi(\mathcal{D})$ the following rule:

$$\leftarrow o(a, T), not\ h(\overline{p}, T) \tag{9}$$

This rule states that a cannot occur if the condition p possibly holds.

4. The inertial law is encoded as follows:

$$ph(L, T + 1) \leftarrow not\ h(\overline{L}, T), not\ dc(\overline{L}, T + 1) \tag{10}$$

$$h(L, T) \leftarrow not\ ph(\overline{L}, T), T \neq 0 \tag{11}$$

which says that L holds at the time moment $T > 0$ if its negation cannot possibly hold at T.

[2] Some adjustment to this syntax is needed if one wants to use some of the existing answer set solvers. For instance, since Cmodels does not allow $h(\neg f, T)$ we may replace it with, say, $h(neg(f), T)$. Besides, to simplify our representation, we make use of choice rules introduced in [24].

5. *Auxiliary rules:* $\pi(\mathcal{D})$ also contains the following rules:

$$\leftarrow h(F,T), h(\neg F,T) \tag{12}$$
$$literal(F) \leftarrow fluent(F) \tag{13}$$
$$literal(\neg F) \leftarrow fluent(F) \tag{14}$$
$$contrary(F, \neg F) \leftarrow fluent(F) \tag{15}$$
$$contrary(\neg F, F) \leftarrow fluent(F) \tag{16}$$

   The first constraint guarantees that two contrary literals cannot hold at the same time. The last four rules are used to define fluent literals and complementary literals.

At this point, it is worthwhile to provide the intuition behind the of atoms $dc(l,T)$, $h(l,T)$, and $ph(l,T)$. Let a be an action and s be a partial state. Consider an "one-step" program $\Pi = \pi(\mathcal{D},1) \cup \{h(l,0) \mid l \in s\} \cup \{o(a,0)\}$.

Observe that Definition 1 implies that a literal l belongs to a possible next state if

1. it is an direct effect of a, i.e., $l \in E(a,\sigma_1)$;
2. it holds by inertial, i.e., $l \in (\sigma_1 \cap \sigma_2)$; or,
3. it is an indirect effects of a, i.e., $l \in \sigma_2 \setminus (E(a,\sigma_1) \cup (\sigma_1 \cap \sigma_2))$. In other words, it is *caused* by a static causal law.

Let S_1, S_2, and S_3 denote the three sets of literals corresponding to the above three cases with respect to the partial state s. Since s might be incomplete, these three sets cannot be computed in full. Our approach is to conservatively estimate the next partial state by

(i) underestimate S_1 by considering only what *definitely will hold given* s. This set is encoded by the set of atoms of the form $dc(l,1)$ and is computed by the rule (5);
(ii) overestimate the negation of S_2 by considering what *can possibly hold in the next state*. This set is encoded by the set of atoms of the form $ph(l,1)$ and is computed by the rules (6) and (8); and
(iii) underestimate S_3 by considering only what *definitely will hold* and what *cannot possibly change* in the construction of the next state. This is encoded by the rules (10)-(11) and (7).

Definition 3. *Let $T^{lp}(\mathcal{D})$ be a transition diagram such that $\langle s,a,s'\rangle \in T^{lp}(\mathcal{D})$ iff s is a partial state and $s' = \{l \mid h(l,1) \in \mathcal{A}\}$ where $\mathcal{A}$ is the answer set of $\pi(\mathcal{D},1) \cup h(s,0) \cup \{o(a,0)\}$.*

The following theorem[3] show that $T^{lp}(\mathcal{D})$ is sound with respect to $T(\mathcal{D})$.

Theorem 1 (Soundness). *If $\mathcal{D}$ is consistent then $T^{lp}(\mathcal{D})$ is a deterministic approximation of $T(\mathcal{D})$.*

4 Approximation Based Conformant Planners

We will now turn our attention to the conformant planning problem in action theories of $\mathcal{AL}$. We begin with the definition of a planning problem.

[3] Proofs of theorems are omitted to save space.

Definition 4. A *planning problem* is a tuple $\langle \mathcal{D}, s^0, s^f \rangle$ where s^0 and s^f are partial states of $\mathcal{D}$.

Partial states s^0 and s^f characterize possible initial situations and the goal respectively.

Definition 5. A chain of events $\alpha = \langle a_0, \ldots, a_{n-1} \rangle$ is a *solution* to a planning problem $\mathcal{P} = \langle \mathcal{D}, s^0, s^f \rangle$ if α is safe in s^0, and for every model M of α with a possible initial state $\sigma_0 \in comp(s^0)$, $M \models s^f$.

We often refer to α as a plan for s^f. If s^0 is a state and action description $\mathcal{D}$ is deterministic then α is a "classical" plan, otherwise it is a *conformant plan*. We next illustrate these definitions using the well-known bomb-in-the-toilet example.

Example 1 (Bomb in the toilet). There is a finite set of toilets and a finite set of packages. One of the packages contains a bomb. The bomb can be disarmed by dunking the package that contains it in a toilet. Dunking a package clogs the toilet. Flushing a toilet unclogs it. Packages can only be dunked in unclogged toilets, one package per toilet. The objective is to find a plan to disarm the bomb. This domain can be modeled by the action description $\mathcal{D}_1$ which consists of the following laws:

$dunk(P, E)$ **causes** $\neg armed(P)$
$dunk(P, E)$ **causes** $clogged(E)$
$flush(E)$ **causes** $\neg clogged(E)$

impossible $dunk(P, E)$ **if** $clogged(E)$
impossible $\{dunk(P, E), flush(E)\}$
impossible $\{dunk(P_1, E), dunk(P_2, E)\}$
impossible $\{dunk(P, E_1), dunk(P, E_2)\}$

E and P are variables for toilets and packages respectively; E_1 and E_2 stand for different toilets and P_1 and P_2 stand for different packages. Note that the last three statements specify physical impossibilities of some concurrent actions and the domain does not have a static causal law.

Let n and m denote the number of packages and toilets respectively. A planning problem in this domain, denoted by $BMTC(n, m)$, is often given by $\langle \mathcal{D}_1, s^0, s^f \rangle$ where s^0 is a (possibly empty) collection of literals of the form $\neg armed(P)$, where P denotes some package. The goal s^f contains $\{\neg armed(1), \ldots, \neg armed(n)\}$.

Consider the problem $BMTC(2, 1)$. We can easily check that if σ is a state containing $\neg clogged(1)$ then $\langle \sigma, dunk(1, 1), \sigma' \rangle$ is a transition in $T(\mathcal{D}_1)$ where $\sigma' = (\sigma \setminus \{armed(1), \neg clogged(1)\}) \cup \{\neg armed(1), clogged(1)\}$. Furthermore,

$$\alpha = \langle flush(1), dunk(1, 1), flush(1), dunk(2, 1) \rangle$$

is safe in the partial state $\emptyset$ and α is a solution to the problem $BMTC(2, 1)$. □

It is not difficult to show that there is a close relationship between conformant plans and paths of an approximation $T'(\mathcal{D})$ of $T(\mathcal{D})$. Because of the soundness of an approximation, it follows from Observation 1 that if $\langle s, \alpha, s' \rangle \in T'(\mathcal{D})$, $s \subseteq s^0$, and $s^f \subseteq s'$ then α is a safe solution in s^0 of the planning problem $\langle \mathcal{D}, s^0, s^f \rangle$.

Since $T^{lp}(\mathcal{D})$ is an approximation of $T(\mathcal{D})$, we can use the program $\pi(\mathcal{D})$ to compute safe solutions of the planning problem $\mathcal{P} = \langle \mathcal{D}, s^0, s^f \rangle$. Furthermore, because $T^{lp}(\mathcal{D})$ is deterministic and computing the next state can be done in polynomial time, we can show that the complexity of the conformant planning problem with respect to $T^{lp}(\mathcal{D})$ is reduced to **NP**-complete (comparing to Σ_2^P, see [30]).

We will next describe the program $\pi(\mathcal{P})$ for this purpose. Like $\pi(\mathcal{D})$, the signature of $\pi(\mathcal{P})$ includes terms corresponding to fluent literals and actions of $\mathcal{D}$. We add to $\pi(\mathcal{P})$ a constant, $length$, which represents the plan length, i.e., time steps can take value in the interval $[0, length]$. We also write $\pi(\mathcal{P}, n)$ to denote the program $\pi(\mathcal{P})$ with $length$ equal to n. $\pi(\mathcal{P})$ consists of $\pi(\mathcal{D})$ and the following rules:

1. *Rules encoding the initial state:* for each $l \in s^0$, we add to $\pi(\mathcal{P})$ the rule:

$$h(l, 0) \leftarrow \tag{17}$$

2. *Goal encoding:* for each $l \in s^f$, $\pi(\mathcal{P})$ contains the constraint:

$$\leftarrow not\ h(l, length)$$

   This set of constraints makes sure that every literal in s^f holds in the final state.
3. *Action generation rule:* as in other ASP-planners, $\pi(\mathcal{P})$ contains the rule for generating action occurrences:

$$1\{o(A, T) : action(A)\} \leftarrow T < length \tag{18}$$

   which says that at each moment of time T, some action must occur[4].

With the help of Theorem 1, we can prove the correctness of the planner $\pi(\mathcal{P})$.

Theorem 2. *Let $\mathcal{A}$ be an answer set of $\pi(\mathcal{P}, n)$. It holds that*

- *for every $0 \leq i < n$, if $a_i = \{e \mid o(e, i) \in \mathcal{A}\}$ then a_i is an action which is not prohibited in $\{l \mid h(l, i) \in \mathcal{A}\}$; and*
- *$\alpha = \langle a_0, \ldots, a_{n-1} \rangle$ is a solution to $\mathcal{P}$.*

This theorem allows us to use $\pi(\mathcal{P})$ for computing minimal plans of $\mathcal{P}$. This is done by sequentially computing the answer sets of $\pi(\mathcal{P}, 0), \pi(\mathcal{P}, 1), \ldots$. In the next section, we will describe our experiments with $\pi(\mathcal{P})$. From now on, we will refer to $\pi(\mathcal{P})$ as CPASP[5]. Before going on, we would like to mention that $\pi(\mathcal{P})$ is not complete. One of the main reasons for the incompleteness of $\pi(\mathcal{P})$ lies in its limited capability in reasoning-by-cases. The next example demonstrates this issue.

Example 2. Consider the action description $\mathcal{D}_2$ consisting of two dynamic causal laws

$$a \textbf{ causes } f \textbf{ if } g \qquad a \textbf{ causes } f \textbf{ if } \neg g$$

Intuitively, we have that a is a conformant plan achieving f from $\emptyset$ because either g or $\neg g$ is true in any state belonging to $comp(\emptyset)$. Yet, it is easy to verify that a cannot be generated by CPASP due to the fact that $\pi(\mathcal{D}_2, 1) \cup h(\emptyset, 0) \cup \{o(a, 0)\}$ has a unique answer set containing no atom of the form $h(l, 1)$. □

The next example shows that it is not only conditional effects but also static causal laws can cause T^{lp} to be incomplete.

[4] If we wish to find a sequential plan, the only thing needed to do is to change the left side of the rule to $1\{o(A, T) : action(A)\}1$.

[5] CPASP stands for **C**onformant **P**lanning using **A**nswer **S**et **P**rogramming.

Example 3. Consider the action description $\mathcal{D}_3$ consisting of the following laws

a **causes** f $\quad$ g **if** f, h $\quad$ g **if** $f, \neg h$

We can check that a is a solution to the problem $\mathcal{P}_3 = \langle \mathcal{D}_3, \{\neg f, \neg g\}, \{g\}\rangle$ since a causes f to hold and the two static causal laws guarantee that if f holds then so does g. Yet, neither $h(h, 1)$ nor $h(\neg h, 1)$ will belong to any answer set of $\pi(\mathcal{P}_3, 1)$ due to the rules (10) and (11). As such, $\pi(\mathcal{P}_3, 1)$ does not return an answer set, i.e., a cannot be found using $T^{lp}(\mathcal{D}_3)$. □

5 Experiments

We ran CPASP on both SMODELS and Cmodels [16]. In general, Cmodels yields better performance. The results reported in this paper are the times obtained using Cmodels. Since most answer set solvers do not scale up well to programs that require large grounded representation, we also implemented the approximation in a C++ planner, called CPA^{ph}([28]). CPA^{ph} employs a best-first search strategy with the number of fulfilled subgoals as its heuristic function. Unlike CPASP, the current version of CPA^{ph} does not compute concurrent plans. However, CPA^{ph} allows disjunctions to be specified in the initial state description, while CPASP does not. Thus, CPASP cannot solve conformant planning benchmarks in the literature where the initial state specification contains disjunctions. We consider this as one of the weaknesses of CPASP.

We compare CPASP (and CPA^{ph}) with three other conformant planners CMBP[9], DLV^k[12], and $\mathcal{C}$-PLAN[8] because these planners do allow static causal laws and are similar in spirit of CPASP (that is, a planning problem is translated into an equivalent problem in a more general setting which can be solved by an off-the-shelf software system). While the latter two allow concurrent planning, the former does not. A comparison between DLV^K and other planners like SGP [25] and GPT [5] can be found in [12]. For a comparison between CPA^{ph} and other state-of-the-art conformant planners like Conformant-FF [6], KACMBP [10], and POND [7], we refer the reader to [28].

We prepared two test suites: one contains sequential, conformant planning benchmarks and the other contains concurrent, conformant planning benchmarks.

The first test suite includes two typical planning domains, the well-known Bomb-in-the-toilet and the $Ring$ domains [10]. In the former, we consider two variants, $BMT(n, p)$ and $BMTC(n, p)$, where n and p are the numbers of packages and toilets respectively. The first one is without clogging and the second one is with clogging. The uncertainty in the initial state is that we do not know whether or not packages are disarmed. In the Ring domain, one can move in a cyclic fashion (either forward or backward) around a n-room building to lock windows. Each room has a window and the window can be locked only if it is closed. Initially, the robot is in the first room and it does not know the state (open/closed) of the windows. The goal is to have all windows locked. A possible conformant plan is to perform a sequence of actions *forward, close, lock* repeatedly. In this domain, we tested with $n \in \{2, 4, 6, 8, 10\}$.

These domains, however, do not contain many static causal laws. Therefore, we introduce two new domains, called $Domino$ and $Gaspipe$. The former is very simple. We have n dominos standing on a line in such a way that if one of them falls then the domino on its right also falls. There is a ball hanging close to the leftmost one. Touching

the ball causes the first domino to fall. Initially, the states of dominos are unknown. The goal is to have the rightmost one to fall. The solution is obviously to touch the ball. In this domain, we tested with $n \in \{100, 200, 500, 100, 2000, 5000, 10000\}$.

The $Gaspipe$ domain is a little more complicated. We need to start a flame in a burner, which is connected to a gas tank through a pipe line. The gas tank is on the left-most of the pipeline and the burner is on the right-most. The pipe line contains sections that connect with each other by valves. The state of pipe sections can be either pressured or unpressured. Opening a valve causes the section on its right side to be pressured if the section on its left is pressured. Moreover, to be safe, a valve can be opened only if the next valve on the line is closed. Closing a valve causes the pipe section on its right side to be unpressured. There are two kinds of static causal laws. The first one is that if a valve is open and the section on its left is pressured then the section on its right will pressured. Otherwise (either the valve is closed or the section on the left is unpressured), the pipe on the right side is unpressured. The burner will start a flame if the pipe connecting to it is pressured. The gas tank is always pressured. The uncertainty we introduce with the initial situation is that the states of valves are unknown. A possible conformant plan will be closing all valves but the first one (that is, the one that connects to the gas tank), in the right-to-left order and then opening them in the reverse order. We tested with $n \in \{3, 5, 7, 9, 11\}$.

The last domain in the first test suite is the $Cleaner$ domain. It is a modified version of the Ring domain. The difference is that instead of locking the window, the robot has to clean objects. Each room has p objects to be cleaned. Initially, the robot is at the first room and does not know whether or not objects are cleaned. The goal is to have all objects cleaned. While the Domino and Gaspipe domains expose a richness in static causal laws, the Cleaner domain provides a high degree of uncertainty in the initial state. We tested the domain with 6 problems where $n \in \{2, 5\}$ and $p \in \{10, 50, 100\}$ respectively.

The second test suite includes benchmarks for concurrent, conformant planning. It contains four domains. The BMT^p and $BMTC^p$ domains are variants of BMT and $BMTC$ in the first test suite in which dunking different packages into different toilets at the same time is allowed. The $Gaspipe^p$ is a modification of $Gaspipe$ in which closing multiple valves at the time are allowed. In addition, one can open a valve while closing other valves. However, it is not allowed to open and close the same valve or open two different valves at the same time. The $Cleaner$ domain is relaxed to allow cleaning multiple objects in the same room at the same time. The relaxed version is denoted by $Cleaner^p$. The testing problems in the second test suite are the same as those in the first test suite.

All experiments were made on a 2.4 GHz CPU, 768MB RAM machine, running Slackware 10.0 operating system. Time limit is set to half an hour. The testing results for two test suites are shown in Tables 1a) and 1b) respectively. We did not test $\mathcal{C}$-PLAN in the sequential planning benchmarks since it is supposed to use for concurrent planning[6]. Times are shown in seconds; "PL", "TO", "MEM", "NA" indicate the length of the plan found by the planner, that the planner ran out of time, that the planner ran out of memory, and that the planner returns a message indicating that no plan can be found[7], respectively. Since both DLV^K and CPASP require as an input parameter the length of

[6] The authors told us that $\mathcal{C}$-PLAN was not intended for searching sequential plans.

[7] We did contact the authors' of the planner for help and are waiting for a response.

Table 1. Comparison between CPASP, CPA^{ph}, CMBP DLV^{K}, and $\mathcal{C}$-PLAN in sequential DLV^{K}, and $\mathcal{C}$-PLAN in sequential a) Sequential Benchmarks b) Concurrent Benchmarks

Domains	CMBP		DLV^{K}		CPASP		CPA^{ph}	
Problems	PL	Time	PL	Time	PL	Time	PL	Time
$BMT(2,2)$	2	0.03	2	0.046	2	0.209	2	0.000
$BMT(4,2)$	4	0.167	4	0.555	4	0.418	4	0.002
$BMT(6,2)$	6	0.206	6	216.557	6	0.775	6	0.005
$BMT(8,4)$	8	0.633		TO	8	6.734	8	0.021
$BMT(10,4)$	10	1.5		TO	10	890.064	10	0.038
$BMTC(2,2)$	2	0.166	2	0.121	2	0.222	2	0.001
$BMTC(4,2)$	6	0.269	6	72.442	6	0.712	6	0.004
$BMTC(6,2)$	10	0.749		TO	8	2.728	10	0.010
$BMTC(8,4)$		TO		TO		TO	12	0.031
$BMTC(10,4)$		TO		TO		TO	16	0.054
$Gaspipe(3)$		NA	5	0.132	5	1.349	7	0.026
$Gaspipe(5)$		NA	9	0.425	9	2.226	22	0.481
$Gaspipe(7)$		NA	13	42.625	13	6.186	86	8.464
$Gaspipe(9)$		NA		TO	17	39.323	261	45.910
$Gaspipe(11)$		NA		TO	21	868.102	1327	529.469
$Cleaner(2,2)$	5	0.1	5	0.104	5	0.496	5	0.002
$Cleaner(2,5)$	11	0.617	11	214.696	11	3.88	11	0.012
$Cleaner(2,10)$		TO		TO		TO	21	0.060
$Cleaner(4,2)$	11	0.13	11	14.82	11	2.094	11	0.014
$Cleaner(4,5)$		TO		TO		TO	23	0.082
$Cleaner(4,10)$		TO		TO		TO	43	0.434
$Cleaner(6,2)$	17	4.1		TO	17	224.391	17	0.054
$Cleaner(6,5)$		TO		TO		TO	35	0.311
$Cleaner(6,10)$		TO		TO		TO	65	1.623
$Ring(2)$	5	0.01		0.201	5	0.911	5	0.003
$Ring(4)$	11	0.116		0.638	11	2.738	12	0.025
$Ring(6)$	17	0.5		TO	17	18.852	18	0.088
$Ring(8)$		TO		TO	23	669.321	24	0.242
$Ring(10)$		TO		TO		TO	30	0.542
$Domino(100)$	1	0.26	1	0.1	1	0.216	1	0.026
$Domino(200)$	1	1.79	1	0.352	1	0.285	1	0.099
$Domino(500)$	1	7.92	1	2.401	1	0.747	1	0.568
$Domino(1000)$	1	13.2	1	13.104	1	1.236	1	2.313
$Domino(2000)$	1	66.6	1	62.421	1	2.414	1	9.209
$Domino(5000)$	1	559.467		MEM	1	6.076	1	67.619
$Domino(10000)$		TO		MEM	1	12.584	1	350.129

a)

Domains	$\mathcal{C}$-PLAN		DLV^{K}		CPASP	
Problems	PL	Time	PL	Time	PL	Time
$BMT^{P}(2,2)$	1	0.078	1	0.074	1	0.116
$BMT^{P}(4,2)$	2	0.052	2	0.094	2	0.268
$BMT^{P}(6,2)$	3	1.812	3	3.065	3	0.346
$BMT^{P}(8,4)$	2	4.32	2	10.529	2	0.248
$BMT^{P}(10,4)$		TO		TO	3	1.911
$BMTC^{P}(2,2)$	1	0.057	1	0.059	1	0.13
$BMTC^{P}(4,2)$	3	0.076	3	0.908	3	0.3
$BMTC^{P}(6,2)$	5	7.519	5	333.278	5	0.672
$BMTC^{P}(8,4)$		TO		TO	3	0.508
$BMTC^{P}(10,4)$		TO		TO	5	1192.458
$Gaspipe^{P}(3)$		TO	4	0.088	4	0.402
$Gaspipe^{P}(5)$		TO	6	0.173	6	0.759
$Gaspipe^{P}(7)$		TO	8	0.441	8	1.221
$Gaspipe^{P}(9)$		TO	10	17.449	10	3.175
$Gaspipe^{P}(11)$		TO		TO	12	8.832
$Cleaner^{P}(2,2)$	3	0.052	3	0.076	3	0.265
$Cleaner^{P}(2,5)$	3	0.121	3	0.066	3	0.3
$Cleaner^{P}(2,10)$	3	0.06	3	0.076	3	0.309
$Cleaner^{P}(4,2)$	7	0.068	7	0.196	7	0.773
$Cleaner^{P}(4,5)$	7	0.09	7	0.809	7	0.931
$Cleaner^{P}(4,10)$	7	0.131	7	237.637	7	1.164
$Cleaner^{P}(6,2)$	11	0.116	11	4.475	11	1.982
$Cleaner^{P}(6,5)$	11	0.195	11	986.731	11	2.947
$Cleaner^{P}(6,10)$	11	0.357		TO	11	3.737

b)

a plan to search for, we ran them by incrementally increasing the plan length, starting from 1[8], until a plan is found.

As can be seen in Table 1a), in the BMT and $BMTC$ domains, CMBP outperforms both DLV^{K} and CPASP in most problems. However, its performance is not competitive with CPA^{ph} which can solve the $BMTC(10,4)$ with only less than one tenth of a second (In fact, CPA^{ph} can scale up to larger problems, e.g., with 100 packages and 100 toilets, within the time limit). CPASP in general has better performance than DLV^{K} in these domains. As an example, DLV^{K} took more than three minutes to solve the $BMT(6,2)$, while it took only 0.775 seconds for CPASP to solve the same problem. Within the time limit, CPASP is able to solve more problems than DLV^{K}.

CPASP seems to work well with domains rich in static causal laws like $Domino$ and $Gaspipe$. In the $Domino$ domain, CPASP outperforms all the other planners in most of instances. It took only 2.414 seconds to solve $Domino(2000)$, while both DLV^{K} and CMBP took more than one minute. Although CPA^{ph} can solve all the instances in this domain, its performance is in general worse than CPASP's. In the $Gaspipe$ domain, CPASP and CPA^{ph} are competitive with each other and outperform the other two. The $Cleaner$ domain turns out to be quite hard for the tested planners except CPA^{ph}. We believe that the high degree of uncertainty in the initial state is the main reason for this performance gain of CPA^{ph}comparing to others since it does not consider *all possible cases* in searching for a solution.

CPASP is outperformed by both CMBP and DLV^{K} in some small instances in the $Ring$ domain. However, it can solve the $Ring(8)$, while CMBP and DLV^{K} cannot.

[8] We did not start from 0 because none of the benchmarks has a plan of length 0.

Again, CPA^{ph} is the best. This shows that CPASP can be competitive with the tested conformant planners in some sequential planning benchmarks.

Table 1b) shows that CPASP also has a fairly good performance in concurrent planning problems. It outperforms both DLV^K and $\mathcal{C}$-PLAN in most instances in the BMT^p, $BMTC^p$, and $Gaspipe^p$ domains. DLV^K is better than $\mathcal{C}$-PLAN in the $Gaspipe^p$ domain. On the contrary, $\mathcal{C}$-PLAN is very good at the $Cleaner^p$ domain. To solve $Cleaner$ $(6, 10)$, $\mathcal{C}$-PLAN took only 0.357 seconds , whereas DLV^K ran out of time and CPASP needs 3.737 seconds.

6 Conclusion and Future Work

We present a logic programming based approximation for $\mathcal{AL}$ action descriptions and apply it to conformant planning. We describe two conformant planners, CPASP and CPA^{ph}, whose key reasoning part is for computing the approximation. Our initial experiments show that with an appropriate approximation, logic programming based conformant planners can be built to deal with problems rich in static causal laws and incomplete information about the initial state. In other words, a careful study in approximated reasoning may pay off well in the development of practical planners.

As an approximation can only guarantee soundness, it will be interesting to characterize situations when an approximation (e.g. $T^{lp}(\mathcal{D})$) can yield completeness. For example, if $T^{lp}(\mathcal{D})$ can generate all conformant plans of length 1 and whenever $\langle a_1, \ldots, a_n \rangle$ is a solution to $\langle \mathcal{D}, s, s^f \rangle$, $\langle a_2, \ldots, a_n \rangle$ is a solution to $\langle \mathcal{D}, s', s^f \rangle$ where $s' = \bigcap_{\exists \sigma \in comp(s). \langle \sigma, a_1, \sigma' \rangle \in T(\mathcal{D})} \sigma'$, then $T^{lp}(\mathcal{D})$ is complete. It can be shown that the first condition can be met when $\mathcal{D}$ does not contain *(i)* a static causal law; *(ii)* a pair of dynamic causal laws of the form a **causes** f **if** p and a **causes** f **if** p' with $p' \cap \bar{p} \neq \emptyset$; and *(iii)* a pair of impossibility conditions of the form **impossible** a **if** p and **impossible** a **if** p' with $p' \cap \bar{p} \neq \emptyset$. Identifying sufficient conditions for the completeness of $T^{lp}(\mathcal{D})$ will be our main concern in the near future.

At this point, we would like to mention that to verify that our approach can deal with a broad spectrum of planning problems, we tested CPASP and CPA^{ph} with several benchmarks problems [1] including the instances of the Blocks World domain tested in [12] and did not encounter a problem that the two planners cannot solve. This shows that our approach can deal with a large class of practical planning problems. Finally, we would like to point out that the use of logic programming allows us to easily exploit control knowledge (e.g., "*do not dunk a package unless it is armed*") in improving the quality of a plan or to specify complex initial (incomplete)-states (see e.g., [13,27]).

Acknowledgment: Michael Gelfond was partially supported by an ARDA contract. Tran Cao Son and Phan Huy Tu were partially supported by NSF grants EIA-0220590 and HRD-0420407.

References

1. F. Bacchus. The AIPS'00 Planning Competition. *AI Magazine*, 22(3), 2001.
2. C. Baral. Reasoning about Actions : Non-deterministic effects, Constraints and Qualification. In *IJCAI'95*, 2017–2023.

3. C. Baral, V. Kreinovich, and R. Trejo. Computational complexity of planning and approximate planning in the presence of incompleteness. *Artificial Intelligence*, 122:241–267, 2000.
4. C. Baral and M. Gelfond. Reasoning agents in dynamic domains. In J. Minker, editor, *Logic-Based Artificial Intelligence*, pages 257–279. Kluwer Academic Publishers, 2000.
5. B. Bonet and H. Geffner. GPT: a tool for planning with uncertainty and partial information. In *IJCAI-01 Workshop on Planning with Uncertainty and Partial Information*, 82–87, 2001.
6. R. Brafman and J. Hoffmann. Conformant planning via heuristic forward search: A new approach. In *(ICAPS-04)*, 355–364.
7. D. Bryce and S. Kambhampati. Heuristic Guidance Measures for Conformant Planning. In *(ICAPS-04)*, 365–375, 2004.
8. C. Castellini, E. Giunchiglia, and A. Tacchella. SAT-based Planning in Complex Domains: Concurrency, Constraints and Nondeterminism. *Artificial Intelligence*, 147:85–117, 2003.
9. A. Cimatti and M. Roveri. Conformant Planning via Symbolic Model Checking. *Journal of Artificial Intelligence Research*, 13:305–338, 2000.
10. A. Cimatti, M. Roveri, and P. Bertoli. Conformant Planning via Symbolic Model Checking and Heuristic Search. *Artificial Intelligence Journal*, 159:127–206, 2004.
11. S. Edelkamp, J. Hoffmann, M. Littman, and H. Younes. The IPC-2004 Planning Competition, 2004. `http://ls5-www.cs.uni-dortmund.de/~edelkamp/ipc-4/`.
12. T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres. A Logic Programming Approach to Knowledge State Planning, II: The DLV$^{\mathcal{K}}$ System. *AIJ*, 144(1-2):157–211, 2003.
13. M. Gelfond and R. Morales. Encoding conformant planning in a-prolog. DRT'04.
14. M. Ghallab et al. PDDL — the Planning Domain Definition Language. Version 1.2. Technical Report CVC TR98003/DCS TR1165, Yale Center for Comp, Vis and Ctrl, 1998.
15. E. Giunchiglia, G. Kartha, and V. Lifschitz. Representing action: indeterminacy and ramifications. *Artificial Intelligence*, 95:409–443, 1997.
16. Y. Lierler and M. Maratea. Cmodels-2: SAT-based Answer Set Solver Enhanced to Non-tight Programs. In LPNMR'04, 346–350, LNCS 2923, 2004.
17. V. Lifschitz. On the Logic of Causal Explanation (Research Note). *AIJ*, 96(2):451–465.
18. F. Lin. Embracing causality in specifying the indirect effects of actions. IJCAI'95, 1985–93.
19. D. Long and M. Fox. The 3rd International Planning Competition: Results and Analysis. JAIR, 20:1–59, 2003.
20. N. McCain & H. Turner. A causal theory of ramifications and qualifications. IJCAI'95, 1978–1984.
21. N. McCain & M. Turner. Causal theories of action and change. AAAI'97, 460–467.
22. S. McIlraith. Intergrating actions and state constraints: A closed-form solution to the ramification problem (sometimes). *Artificial Intelligence*, 116:87–121, 2000.
23. M. Shanahan. The ramification problem in the event calculus. In IJCAI'99, 140–146, 1999.
24. P. Simons, N. Niemelä, and T. Soininen. Extending and Implementing the Stable Model Semantics. *Artificial Intelligence*, 138(1–2):181–234, 2002.
25. D. Smith and D. Weld. Conformant graphplan. In *Proceedings of AAAI 98*, 1998.
26. T.C. Son and C. Baral. Formalizing sensing actions - a transition function based approach. *Artificial Intelligence*, 125(1-2):19–91, January 2001.
27. T.C. Son, C. Baral, N. Tran, and S. McIlraith. Domain-Dependent Knowledge in Answer Set Planning. To appear in TOCL.
28. T.C. Son, P.H. Tu, M. Gelfond, and R. Morales. Conformant Planning for Domains with Constraints — A New Approach. To Appear in AAAI'05.
29. S. Thiebaux, J. Hoffmann, and B. Nebel. In Defense of PDDL Axioms. IJCAI'03.
30. H. Turner. Polynomial-length planning spans the polynomial hierarchy. JELIA'02, 111–124.

Game-Theoretic Reasoning About Actions in Nonmonotonic Causal Theories

Alberto Finzi[1,2] and Thomas Lukasiewicz[1,2]

[1] Institut für Informationssysteme, Technische Universität Wien
Favoritenstraße 9-11, A-1040 Vienna, Austria
{finzi, lukasiewicz}@kr.tuwien.ac.at

[2] Dipartimento di Informatica e Sistemistica, Università di Roma "La Sapienza"
Via Salaria 113, I-00198 Rome, Italy
{finzi, lukasiewicz}@dis.uniroma1.it

Abstract. We present the action language $GC+$ for reasoning about actions in multi-agent systems under probabilistic uncertainty and partial observability, which is an extension of the action language $\mathcal{C}+$ that is inspired by partially observable stochastic games (POSGs). We provide a finite-horizon value iteration for this framework and show that it characterizes finite-horizon Nash equilibria. We also describe how the framework can be implemented on top of nonmonotonic causal theories. We then present acyclic action descriptions in $GC+$ as a special case where transitions are computable in polynomial time. We also give an example that shows the usefulness of our approach in practice.

1 Introduction

There are several important problems that we have to face in reasoning about actions for mobile agents in real-world environments. First and foremost, we have to deal with uncertainty, both about the initial situation of the agent's world and about the results of the actions taken by the agent (due to noisy effectors and/or sensors). Second, a closely related problem is that the properties of real-world environments are in general not fully observable (due to noisy and inaccurate sensors, or because some relevant parts of the environment simply cannot be sensed), and thus we also have to deal with partial observability. One way of adding uncertainty and partial observability to reasoning about actions is based on qualitative models in which all possible alternatives are equally taken into consideration. Another way is based on quantitative models where we have a probability distribution on the set of possible alternatives, and thus can numerically distinguish between the possible alternatives.

Well-known first-order formalisms for reasoning about actions such as the situation calculus [24] easily allow for expressing qualitative uncertainty about the effects of actions and the initial situation of the world through disjunctive knowledge. Furthermore, there are generalizations of the action language $\mathcal{A}$ [12] that allow for qualitative uncertainty in the form of nondeterministic actions. An important recent formalism in this family is the action language $\mathcal{C}+$ [13], which is based on the theory of nonmonotonic causal reasoning presented in [18], and has evolved from the action language $\mathcal{C}$. In

C. Baral et al. (Eds.): LPNMR 2005, LNAI 3662, pp. 185–197, 2005.
© Springer-Verlag Berlin Heidelberg 2005

addition to allowing for conditional and nondeterministic effects of actions, $\mathcal{C}+$ also supports concurrent actions as well as indirect effects and preconditions of actions through static causal laws. Closely related to it is the recent planning language $\mathcal{K}$ [7].

There are a number of formalisms for probabilistic reasoning about actions. In particular, Bacchus et al. [1] propose a probabilistic generalization of the situation calculus, which is based on first-order logics of probability, and which allows to reason about an agent's probabilistic degrees of belief and how these beliefs change when actions are executed. Poole's independent choice logic [22] is based on acyclic logic programs under different "choices". Each choice along with the acyclic logic program produces a first-order model. By placing a probability distribution over the different choices, we then obtain a distribution over the set of first-order models. Boutilier et al. [5] introduce and explore an approach to first-order (fully observable) Markov decision processes (MDPs) [23] that are formulated in a probabilistic generalization of the situation calculus. A companion paper [6] presents a generalization of Golog, called DTGolog, that combines agent programming in Golog with decision-theoretic planning in MDPs. Probabilistic extensions of the action language $\mathcal{A}$ and its most recent variant $\mathcal{C}+$ have especially been proposed by Baral et al. [2] and Eiter and Lukasiewicz [8].

Many of the above logical formalisms for reasoning about actions under probabilistic uncertainty take inspiration from decision-theoretic planning in fully observable Markov decision processes (MDPs) [23] and the more general partially observable Markov decision processes (POMDPs) [16]. Such logical formalisms for reasoning about actions that are inspired by decision-theoretic planning are also appealing from the perspective of decision-theoretic planning, since they allow for [11,14] (i) compactly representing MDPs and POMDPs without explicitly referring to atomic states and state transitions, (ii) exploiting such compact representations for efficiently solving large-scale problems, and (iii) nice properties such as *modularity* (parts of the specification can be easily added, removed, or modified) and *elaboration tolerance* (solutions can be easily reused for similar problems with few or no additional effort).

The above generalizations of $\mathcal{A}$ and $\mathcal{C}+$ in [2,8] assume that the model of the world consists of a single agent that we want to control and the environment summarized in "nature". In realistic applications, however, we often encounter multiple agents, which may compete or cooperate with each other. Here, the optimal actions of one agent generally depend on the actions of all the other agents. In particular, there is a bidirectional dependence between the actions of two agents, which generally makes it inappropriate to model enemies and friends of the controlled agent simply as a part of "nature".

There are generalizations of MDPs and POMDPs to multi-agent systems with cooperative agents, called multi-agent MDPs [4] and decentralized POMDPs [3,20], respectively. Similarly, there are also generalizations of MDPs and POMDPs to multi-agent systems with competing (that is, not necessarily cooperative) agents, called stochastic games [21] (or Markov games [25,17]) and partially observable stochastic games (POSGs) [15,9], respectively. Multi-agent MDPs (resp., decentralized POMDPs) and stochastic games (resp., POSGs) are similar to MDPs (resp., POMDPs), except that actions (and decisions) are distributed among multiple agents, where the optimal actions of each agent may depend on the actions of all the other agents. Stochastic games (resp., POSGs) generalize both normal form games [26] and MDPs (resp., POMDPs).

In this paper, we present the language $\mathcal{GC}+$ for reasoning about actions in multi-agent systems under probabilistic uncertainty and partial observability, which is an extension of the language $\mathcal{C}+$ that takes inspirations from partially observable stochastic games (POSGs) [15]. The main contributions of this paper are as follows:

- We present the action language $\mathcal{GC}+$ for reasoning about actions in multi-agent systems under probabilistic uncertainty and partial observability, which is an extension of both the action language $\mathcal{C}+$ and POSGs. We consider the very general case in which the agents may have different rewards, and thus may be competitive. Here, we assume that planning and control are centralized as follows. All agents transmit their local belief states and/or observations to a central agent, which then computes and returns the optimal local action for each agent.
- Under the above assumption, the high worst-case complexity of POSGs (NEXP-completeness for the special case of decentralized POMDPs [3]) is avoided, since the POSG semantics of $\mathcal{GC}+$ can be translated into a belief state stochastic game semantics. We use the latter to define a finite-horizon value iteration for $\mathcal{GC}+$, and show that it characterizes finite-horizon Nash equilibria.
- We show that the $\mathcal{GC}+$ framework can be implemented on top of reasoning in nonmonotonic causal theories. We present acyclic action descriptions in $\mathcal{GC}+$ as a special case where transitions are computable in polynomial time. We also provide an example that shows the usefulness of our approach in practice.

Note that further technical details are given in the extended paper [10].

2 Preliminaries

In this section, we recall the basic concepts of the action language $\mathcal{C}+$, normal form games, and partially observable stochastic games.

2.1 The Action Language $\mathcal{C}+$

We first recall the main concepts of the action language $\mathcal{C}+$; see especially [13] for further details, motivation, and background.

Syntax. Properties of the world are represented by rigid variables, simple fluents, and statically determined fluents, while actions are expressed by action variables. The values of rigid variables do not change when actions are performed, while the ones of simple (resp., statically determined) fluents may directly (resp., indirectly) change through actions. The knowledge about the latter is encoded through dynamic (resp., static) causal laws over formulas, which are Boolean combinations of atomic assignments.

Formally, we thus assume a finite set V of *variables*, which are divided into *rigid variables*, *simple fluents*, *statically determined fluents*, and *action variables*. Every variable $X \in V$ may take on *values* from a nonempty finite *domain* $D(X)$, where every action variable has the Boolean domain $\{\bot, \top\}$. We define *formulas* inductively as follows. *False* and *true*, denoted $\bot$ and $\top$, respectively, are formulas. If $X \in V$ and $x \in D(X)$, then $X = x$ is a formula (called *atom*). If ϕ and ψ are formulas, then also $\neg\phi$ and $(\phi \wedge \psi)$. A *literal* is an atom $X = x$ or a negated atom $\neg X = x$ (abbreviated as $X \neq x$). We often abbreviate $X = \top$ (resp., $X = \bot$) as X (resp., $\neg X$).

Static causal laws express static knowledge about fluents and rigid variables. They are expressions of the form

$$\textbf{caused}\ \psi\ \textbf{if}\ \phi\,, \tag{1}$$

where ψ and ϕ are formulas such that either (a) every variable in ψ is a fluent, and no variable in ϕ is an action variable, or (b) every variable in ψ and ϕ is rigid. Informally, (1) encodes that every state of the world that satisfies ϕ should also satisfy ψ. If $\phi = \top$, then (1) is abbreviated by **caused** ψ. *Dynamic causal laws* express how simple fluents change when actions are performed. They have the form

$$\textbf{caused}\ \psi\ \textbf{if}\ \phi\ \textbf{after}\ \theta\,, \tag{2}$$

where ψ, ϕ, and θ are formulas such that every variable in ψ is a simple fluent, and no variable in ϕ is an action variable. Informally, (2) encodes that every next state of the world satisfying ϕ should also satisfy ψ, if the current state and the executed action satisfy θ. If $\phi = \top$, then (2) is abbreviated by **caused** ϕ **after** θ. If also $\theta = a_1 \wedge \cdots \wedge a_k \wedge \delta$, where every a_i is an assignment of $\top$ to an action variable, then (2) is abbreviated by $a_1, \ldots, a_k$ **causes** ψ **if** δ. Informally, if the current state of the world satisfies δ, then the next state after concurrently executing $a_1, \ldots, a_k$ satisfies ψ. If $\psi = \bot$ and $\phi = \top$, then (2) is an *execution denial* and abbreviated by

$$\textbf{nonexecutable}\ \theta\,. \tag{3}$$

Informally, if a state s and an action α satisfy θ, then α is not executable in s. If $\theta = a_1 \wedge \cdots \wedge a_k \wedge \delta$, then (3) is abbreviated by **nonexecutable** $a_1, \ldots, a_k$ **if** δ. Informally, $a_1, \ldots, a_k$ cannot be concurrently executed in a state satisfying δ. The expression **inertial** X, where $X \in V$, abbreviates the set of all laws (2) such that $\phi = \psi = \theta = X{=}x$ and $x \in D(X)$. Informally, the value of X remains unchanged when actions are executed, as long as this does not produce any inconsistencies.

A *causal law* (or *axiom*) is a static or dynamic causal law. An *action description* D is a finite set of causal laws. An *initial database* ϕ is a formula without action variables.

Semantics. An action description D represents a system of transitions from states to sets of possible successor states, while an initial database ϕ encodes a set of possible initial states. We now define states and actions, the executability of actions in states, and the above transitions through actions.

An *interpretation* I of a set of variables $V' \subseteq V$ assigns to every $X \in V'$ an element of $D(X)$. We say I *satisfies* an atom $Y = y$, where $Y \in V'$, denoted $I \models Y = y$, iff $I(Y) = y$. Satisfaction is extended to all formulas over V' as usual.

Let s be an interpretation of all rigid variables and fluents in V. Let D^s be the set of all ψ such that either (a) $s \models \phi$ for some **caused** ψ **if** ϕ in D, or (b) $s \models \psi$ and $\psi = X{=}x$ for some simple fluent $X \in \mathcal{X}$ and $x \in D(X)$. A *state* s of D is an interpretation s as above that is a unique model of D^s. An *action* α is an interpretation of all action variables in V. The action α is *executable* in a state s, denoted $Poss(\alpha, s)$, iff $s \cup \alpha$ satisfies $\neg\theta$ for every **nonexecutable** θ in D.

An *action transition* is a triple (s, α, s'), where s and s' are states of D such that $s(X) = s'(X)$ for every rigid variable $X \in V$, and α is an action that is executable in s. A formula ψ is *caused* in (s, α, s') iff either (a) $s' \models \phi$ for some **caused** ψ **if** ϕ in D, or (b) $s \cup \alpha \models \theta$ and $s' \models \phi$ for some **caused** ψ **if** ϕ **after** θ in D. The triple (s, α, s') is *causally explained* iff s' is the only interpretation that satisfies all formulas caused

in (s, α, s'). For every state s and action α, define $\Phi(s, \alpha)$ as the set of all states s' such that (s, α, s') is *causally explained*. Note that $\Phi(s, \alpha) = \emptyset$ if no such (s, α, s') exists, in particular, if α is not executable in s. We say that D is *consistent* iff $\Phi(s, \alpha) \neq \emptyset$ for all actions α and states s such that α is executable in s. Informally, $\Phi(s, \alpha)$ is the set of all possible successor states after executing α in s.

2.2 Normal Form Games

Normal form games from classical game theory [26] describe the possible actions of $n \geqslant 2$ agents and the rewards (or utilities) that the agents receive when they simultaneously execute one action each. For example, in *two-finger Morra*, two players E and O simultaneously show one or two fingers. Let f be the total numbers of fingers shown. If f is odd, then O gets f dollars from E, and if f is even, then E gets f dollars from O. Formally, a *normal form game* $G = (I, (A_i)_{i \in I}, (R_i)_{i \in I})$ consists of a set of *agents* $I = \{1, \ldots, n\}$, $n \geqslant 2$, a nonempty finite set of *actions* A_i for each agent $i \in I$, and a *reward* (or *utility*) *function* $R_i \colon A \to \mathbf{R}$ for each agent $i \in I$, which associates with every *joint action* $a \in A = \times_{i \in I} A_i$ a *reward* (or *utility*) $R_i(a)$ to agent i.

A pure (resp., mixed) strategy specifies which action an agent should execute (resp., which actions an agent should execute with which probability). Formally, a *pure strategy* for agent $i \in I$ is any action $a_i \in A_i$. A *pure strategy profile* is any joint action $a \in A$. If the agents play a, then the *reward* to agent $i \in I$ is $R_i(a)$. A *mixed strategy* for agent $i \in I$ is any probability distribution π_i over A_i. A *mixed strategy profile* $\pi = (\pi_i)_{i \in I}$ consists of a mixed strategy π_i for each agent $i \in I$. If the agents play π, then the *expected reward* to agent $i \in I$, denoted $\mathbf{E}[R_i(a) \mid \pi]$ (or $R_i(\pi)$), is defined as

$$\textstyle\sum_{a=(a_j)_{j \in I} \in A} R_i(a) \cdot \Pi_{j \in I} \pi_j(a_j) \, .$$

We are especially interested in mixed strategy profiles π, called Nash equilibria, where no agent has the incentive to deviate from its part, once the other agents play their parts. A mixed strategy profile $\pi = (\pi_i)_{i \in I}$ is a *Nash equilibrium* for G iff for every agent $i \in I$, it holds that $R_i(\pi'_i \circ \pi_{-i}) \leqslant R_i(\pi)$ for every mixed strategy π'_i, where π_{-i} (resp., $\pi'_i \circ \pi_{-i}$) is obtained from π by removing π_i (resp., replacing π_i by π'_i). Every normal form game G has at least one Nash equilibrium among its mixed (but not necessarily pure) strategy profiles, and many have multiple Nash equilibria. A *Nash selection function* f associates with every normal form game G a unique Nash equilibrium $f(G)$. The expected reward to agent $i \in I$ under $f(G)$ is denoted by $v^i_f(G)$.

2.3 Partially Observable Stochastic Games

We will use POSGs [15] to define the semantics of the action language $\mathcal{GC}+$, where we assume that planning and control are centralized as follows. There exists a central agent, which (i) knows the local belief state of every other agent, (ii) computes and sends them their optimal local actions, and (iii) thereafter receives their local observations. Hence, we assume a transmission of local belief states and local observations to a central agent from all other agents, and of the optimal local actions in the reverse direction. Using this assumption, we can translate POSGs into belief state stochastic games, and then perform a finite-horizon value iteration.

Roughly, a POSG consists of a nonempty finite set of states S, a normal form game for each state $s \in S$, a set of joint observations of the agents O, and a transition function that associates with every state $s \in S$ and joint action of the agents $a \in A$ a probability distribution on all combinations of next states $s' \in S$ and joint observations $o \in O$. Formally, a *partially observable stochastic game (POSG)* $G = (I, S, (A_i)_{i \in I}, (O_i)_{i \in I}, P, (R_i)_{i \in I})$ consists of a set of *agents* $I = \{1, \ldots, n\}$, $n \geqslant 2$, a nonempty finite set of *states* S, two nonempty finite sets of *actions* A_i and *observations* O_i for each agent $i \in I$, a transition function $P \colon S \times A \to PD(S \times O)$, which associates with every state $s \in S$ and joint action $a \in A = \times_{i \in I} A_i$ a probability distribution over $S \times O$, where $O = \times_{i \in I} O_i$, and a *reward function* $R_i \colon S \times A \to \mathbf{R}$ for each agent $i \in I$, which associates with every state $s \in S$ and joint action $a \in A$ a *reward* $R_i(s, a)$ to agent i.

Since the actual state $s \in S$ of the POSG G is not fully observable, every agent $i \in I$ has a belief state b_i that associates with every state $s \in S$ the belief of agent i about s being the actual state. Formally, a *belief state* $b = (b_i)_{i \in I}$ of G consists of a probability function b_i over S for each agent $i \in I$. The POSG G then defines probabilistic transitions between belief states as follows. The new belief state $b^{a,o} = (b_i^{a,o})_{i \in I}$ after executing the joint action $a \in A$ in $b = (b_i)_{i \in I}$ and jointly observing $o \in O$ is given by:

$$b_i^{a,o}(s') = \textstyle\sum_{s \in S} P(s', o \,|\, s, a) \cdot b_i(s) \;/\; P_b(b_i^{a,o} \,|\, b_i, a), \text{ where}$$
$$P_b(b_i^{a,o} \,|\, b_i, a) = \textstyle\sum_{s' \in S} \sum_{s \in S} P(s', o \,|\, s, a) \cdot b_i(s)$$

is the probability of observing o after executing a in b_i. These probabilistic transitions define the fully observable stochastic game over belief states $G' = (I, B, (A_i)_{i \in I}, P_b, (R_i)_{i \in I})$, where B is the set of all belief states of G.

We next define finite-horizon pure and mixed policies and their rewards and expected rewards, respectively, using the above fully observable stochastic game over belief states. Assuming a finite horizon $H \geqslant 0$, a pure (resp., mixed) time-dependent policy associates with every belief state b of G and number of steps to go $h \in \{0, \ldots, H\}$ a pure (resp., mixed) normal form game strategy. Formally, a *pure policy* α assigns to each belief state b and number of steps to go $h \in \{0, \ldots, H\}$ a joint action from A. A *mixed policy* is of the form $\pi = (\pi_i)_{i \in I}$, where every π_i assigns to each belief state b and number of steps to go $h \in \{0, \ldots, H\}$ a probability function $\pi_i[b, h]$ over A_i. The H-step reward (resp., expected H-step reward) for pure (resp., mixed) policies can now be defined as usual. In particular, the *expected H-step reward* to agent $i \in I$ under a start belief state $b = (b_i)_{i \in I}$ and the mixed policy π, denoted $G_i(H, b, \pi)$, is defined as

$$\begin{cases} \sum_{a \in A} (\Pi_{j \in I} \pi_j[b, 0](a_j)) \cdot \sum_{s \in S} b_i(s) R_i(s, a) & \text{if } H = 0; \\ \sum_{a \in A} (\Pi_{j \in I} \pi_j[b, H](a_j)) \cdot (\sum_{s \in S} b_i(s) R_i(s, a) + \\ \qquad \sum_{o \in O} P(b_i^{a,o} | b_i, a) \cdot G_i(H{-}1, b^{a,o}, \pi)) & \text{otherwise.} \end{cases}$$

The notion of a finite-horizon Nash equilibrium for a POSG G is then defined as follows. A policy π is a *Nash equilibrium* of G under a belief state b iff for every agent $i \in I$, it holds that $G_i(H, b, \pi'_i \circ \pi_{-i}) \leqslant G_i(H, b, \pi_i \circ \pi_{-i})$ for all policies π'_i. A policy π is a *Nash equilibrium* of G iff it is a Nash equilibrium of G under every belief state b.

Nash equilibria of G can be characterized by finite-horizon value iteration from local Nash equilibria of normal form games as follows. Let f be an arbitrary Nash selection function for normal form games with the action sets $(A_i)_{i \in I}$. For every belief state $b = (b_i)_{i \in I}$ and number of steps to go $h \in \{0, \ldots, H\}$, let $G[b, h] = (I, (A_i)_{i \in I}, (Q_i[b, h])_{i \in I})$, where $Q_i[b, h](a)$ is defined as follows (for all $a \in A$ and $i \in I$):

$$\begin{cases} \sum_{s \in S} b_i(s) R_i(s,a) & \text{if } h = 0; \\ \sum_{s \in S} b_i(s) R_i(s,a) + \sum_{o \in O} P(b_i^{a,o} \,|b_i, a) \cdot v_f^i(G[b^{a,o}, h-1]) & \text{otherwise.} \end{cases}$$

Let the mixed policy $\pi = (\pi_i)_{i \in I}$ for the POSG G be defined by $\pi_i(b,h) = f_i(G[b,h])$ for all $i \in I$, belief states b, and number of steps to go $h \in \{0, \ldots, H\}$. Then, π is a Nash equilibrium of G, and $G_i(H, b, \pi) = v_f^i(G[b, H])$ for every $i \in I$ and belief state b.

3 The Action Language G$\mathcal{C}+$

In this section, we define the action language G$\mathcal{C}+$, which generalizes both the action language $\mathcal{C}+$ and POSGs.

Syntax. We extend $\mathcal{C}+$ by formulas that express probabilistic transitions and agent rewards as in POSGs as well as formulas that encode the initial belief state of the agents.

We assume a set of $n \geqslant 2$ agents $I = \{1, \ldots, n\}$. Each agent $i \in I$ has (i) a nonempty set of action variables AV_i, where $AV_1, \ldots, AV_n$ partitions the set of all action variables $AV \subseteq V$, and (ii) a nonempty set of possible observations O_i. Every $o \in O = \times_{i \in I} O_i$ is a *joint observation*. A *probabilistic dynamic causal law* is of the form

$$\textbf{caused}\ [(\psi_1 \textbf{ if } \phi_1; o_1) : p_1, \ldots, (\psi_k \textbf{ if } \phi_k; o_k) : p_k]\ \textbf{after}\ \delta\,, \tag{4}$$

where every **caused** ψ_j **if** ϕ_j **after** δ with $j \in \{1, \ldots, k\}$ is a dynamic causal law, every o_j is a joint observation, $p_1, \ldots, p_k > 0$, $p_1 + \cdots + p_k = 1$, and $k \geqslant 1$. Informally, if an action α is executed in a state s, where $s \cup \alpha \models \delta$, then with the probability p_j the successor states satisfy **caused** ψ_j **if** ϕ_j and the agents observe o_j. We omit "**if** ϕ_j" in (4), when $\phi_j = \top$. A *reward law* for agent $i \in I$ is of the form

$$\textbf{reward}\ i \colon r\ \textbf{after}\ \delta\,, \tag{5}$$

where r is a real. Informally, if an action α is executed in a state s, where $s \cup \alpha \models \delta$, then agent i receives the reward r. A *probabilistic initial database law* for $i \in I$ is of form

$$i \colon [\psi_1 : p_1, \ldots, \psi_k : p_k]\,, \tag{6}$$

where each ψ_j with $j \in \{1, \ldots, k\}$ is a formula without action variables, $p_1, \ldots, p_k > 0$, $p_1 + \cdots + p_k = 1$, and $k \geqslant 1$. Informally, the initial belief of agent i is that the set of states satisfying ψ_j holds with the probability p_j.

A *probabilistic action description* P is a finite set of causal, probabilistic dynamic causal, and reward laws. A *probabilistic initial database* $\Psi = (\Psi_i)_{i \in I}$ consists of a probabilistic initial database law Ψ_i for every agent $i \in I$.

Semantics. A probabilistic action description P represents a transition system, where every state s and action α executable in s is associated with a reward to every agent and a probability distribution over possible successor states. A probabilistic initial database $\Psi = (\Psi_i)_{i \in I}$ encodes each agent's probabilistic belief about the possible initial states.

The set of all states and actions of P and the executability of an action in a state are defined as in Section 2.1. An *action for agent* $i \in I$ is any interpretation over AV_i. The set of all actions for agent i is denoted by A_i. We next define the probabilistic transitions and the rewards encoded in P.

Let s be a state, and let α be an action executable in s. Suppose that P contains exactly one law F of the form (4) such that $s \cup \alpha \models \delta$. For every $j \in \{1, \ldots, k\}$, let P_j be obtained from P by replacing F by **caused** ψ_j **if** ϕ_j **after** δ. Let $\Phi_j(s, \alpha)$ be the set of all states s' such that (s, α, s') is causally explained relative to P_j. For each state s' and $o \in O$, let $P_j(s', o|s, \alpha) = p_j \,/\, |\Phi_j(s, \alpha)|$, if $s' \in \Phi_j(s, \alpha)$ and $o = o_j$, and $P_j(s', o|s, \alpha) = 0$, otherwise. Informally, p_j is uniformly distributed among all $s' \in \Phi_j(s, \alpha)$. For each state s' and $o \in O$, the probability of moving to the successor state s' along with jointly observing o, when executing α in s, denoted $P(s', o|s, \alpha)$, is defined as $\sum_{j=1}^{k} P_j(s', o|s, \alpha)$.

Let s be a state, and let α be an action executable in s. Suppose for every agent $i \in I$, exactly one law **reward** $i\colon r$ **after** δ with $s \cup \alpha \models \delta$ belongs to P. Then, the *reward* to i when executing α in s, denoted $R_i(s, \alpha)$, is defined as r.

We next define the initial probabilistic belief of every agent $i \in I$, which is encoded in the law Ψ_i of the form (6). For each $j \in \{1, \ldots, k\}$, let Φ_j be the set of all states satisfying ψ_j. For each state s, let $P_j(s) = p_j \,/\, |\Phi_j|$, if $s \in \Phi_j$, and $P_j(s) = 0$, otherwise. Agent i's belief about s being the initial state, denoted $b_i^0(s)$, is defined as $\sum_{j=1}^{k} P_j(s)$.

In the sequel, we implicitly assume that all P and Ψ are consistent: We say that P is *consistent* iff for each state s and action α executable in s, (i) there is exactly one law (4) in P with $s \cup \alpha \models \delta$, (ii) each $\Phi_j(s, \alpha)$ as above is nonempty, and (iii) for every agent $i \in I$, there is exactly one law **reward** $i\colon r$ **after** δ in P with $s \cup \alpha \models \delta$. We say that Ψ is *consistent* iff, for every $i \in I$, each Φ_j as above is nonempty.

Example 3.1 (Two Robots). We consider the scenario shown in Fig. 1: There are two robots a_1 and a_2 in a room looking for an object o_1, and trying to bring it out through the only door d_1. Both robots can pick up the object, and also pass it to another robot. A pass attempt is only possible if the two robots are facing in adjacent positions. If the receiving robot is not expecting the object, then it falls down. If the two robots are in the same location, then they both cannot perform any pick up and door crossing action. We assume that the reward for the robot bringing out the object is a bit higher. Hence, there is an additional individual payoff for the robot able to accomplish the goal.

Let $\mathcal{L} = \{l_{1,1}, \ldots, l_{2,3}, d_1, \mathit{nil}\}$ be the set of possible locations of the robots and the object, where $l_{i,j}$ encodes the field (i, j), and d_1 represents the door. For locations L and L', let $\mathit{close}(L, L')$ be true iff L and L' are adjacent. We assume the simple fluents $at(X)$, where $X \in \{a_1, a_2, o_1\}$, with the domain $\mathcal{L}$, as well as $\mathit{holds}(R)$, where $R \in \{a_1, a_2\}$, with the domain $\{o_1, \mathit{nil}\}$. Let the action variables be given by $\mathit{goTo}(R, L)$, $\mathit{pickUp}(R)$, $\mathit{passTo}(R, R')$, $\mathit{receive}(R)$, where $R, R' \in \{a_1, a_2\}$, $R \neq R'$,

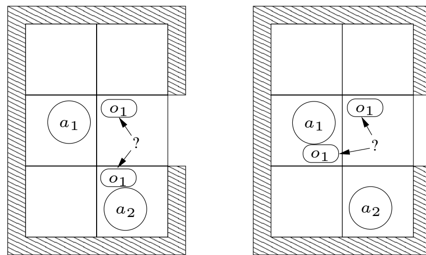


Fig. 1. Initial belief states of a_2 and a_1, respectively

and $L \in \mathcal{L}$. Each robot's set of observations is $\{obs(holds),\ obs(notHolds)\}$. Informally, each robot can only check if it is carrying something or not after a pick up. We assume the following static causal law:

$$\textbf{caused}\ at(O) = nil\ \textbf{if}\ holds(R) = O\,.$$

We introduce the following dynamic causal laws for the action variables $passTo(R,R')$, $receive(R)$, and $goTo(R, L)$ (they abbreviate probabilistic causal laws (4) with $k = 1$):

$$\begin{aligned}
&\textbf{caused}\ holds(R) = nil\ \textbf{after}\ passTo(R, R')\ \text{with}\ R \neq R'\,,\\
&\textbf{caused}\ holds(R) = O\ \textbf{after}\ holds(R') = O \wedge passTo(R', R) \wedge\\
&\qquad receive(R)\ \text{with}\ R \neq R'\,,\\
&\textbf{caused}\ at(O) = L\ \textbf{after}\ holds(R) = O \wedge passTo(R, R') \wedge\\
&\qquad \neg receive(R') \wedge at(R, L)\ \text{with}\ R \neq R'\,,\\
&\textbf{caused}\ at(R) = L\ \textbf{after}\ goTo(R, L)\,.
\end{aligned}$$

Here, if R fails to pass the object O, the latter remains in the location of R. For $pickUp(R)$, we introduce the following probabilistic causal law, assuming $pickUp(R)$ can fail, and $obs(notHolds)$ can give incorrect positive results:

$$\begin{aligned}
\textbf{caused}\ [&(holds(R) = O\ ; obs(holds)) : 0.7,\\
&(holds(R) = O\ ; obs(notHolds)) : 0.1,\\
&(holds(R) = nil\ ; obs(notHolds)) : 0.2]\\
&\qquad \textbf{after}\ pickUp(R) \wedge at(R) = L \wedge at(O) = L\,,
\end{aligned}$$

We assume the following execution denials:

$$\begin{aligned}
&\textbf{nonexecutable}\ pickUp(R) \wedge holds(R) \neq nil\,,\\
&\textbf{nonexecutable}\ pickUp(R) \wedge at(R) = L \wedge at(o_1) \neq L\,,\\
&\textbf{nonexecutable}\ pickUp(R) \wedge at(R') = L \wedge at(R) = L\,,\\
&\textbf{nonexecutable}\ goTo(R, L) \wedge at(R) = L' \wedge \neg close(L, L')\,,\\
&\textbf{nonexecutable}\ goTo(R, d_1) \wedge at(R) = L \wedge at(R') = L\,,\\
&\textbf{nonexecutable}\ passTo(R, R') \wedge at(R){=}L \wedge at(R'){=}L' \wedge \neg close(L, L')\,.
\end{aligned}$$

where $R' \neq R$ and $L' \neq L$. Furthermore, every robot can execute only one action at a time, that is, for any two distinct actions α and α' of either robot a_1 or a_2:

$$\textbf{nonexecutable}\ \alpha \wedge \alpha'\,.$$

For every simple fluent X, we assume the inertial law **inertial** X. Finally, the reward function is defined as follows:

$$\begin{aligned}
&\textbf{reward}\ a_i : 100\ \textbf{after}\ \alpha_i \wedge holds(a_i, O)\,,\\
&\textbf{reward}\ a_i : 90\ \textbf{after}\ \alpha_i \wedge holds(a_j, O)\ \ \text{with}\ i \neq j\,,\\
&\textbf{reward}\ a_i : 10\ \textbf{after}\ \alpha \wedge holds(a_i, O)\ \text{with}\ \alpha \neq \alpha_i\,,\\
&\textbf{reward}\ a_i : 0\ \textbf{after}\ \alpha \wedge \textstyle\bigwedge_{i=1,2} \neg holds(a_i, O)\ \text{with}\ \alpha \neq \alpha_i.
\end{aligned}$$

where $\alpha_i = goTo(a_i, d_1)$. The robot achieving the goal receives a high reward, the other one a bit less. If a robot moves carrying something, it also receives a small payoff.

4 Finite-Horizon Value Iteration

In this section, we define finite-horizon Nash equilibria for probabilistic action descriptions P in $\mathcal{GC}+$ and provide a finite-horizon value iteration for computing them.

Nash Equilibria. We first define belief states and probabilistic transitions between them. A *belief state* of P is of the form $b = (b_i)_{i \in I}$, where every b_i is a probability function over the set of states of P. An action α is *executable* in $b = (b_i)_{i \in I}$ iff for every $i \in I$ the action α is executable in some state s with $b_i(s) > 0$. Then, the new belief state $b^{\alpha,o} = (b_i^{\alpha,o})_{i \in I}$ after executing α in b and observing $o \in O$ is given by:

$$b_i^{\alpha,o}(s') = \sum_{s \in S, Poss(\alpha,s)} P(s', o | s, \alpha) \cdot b_i(s) / P(b_i^{\alpha,o} | b_i, \alpha), \text{ where}$$
$$P(b_i^{\alpha,o} \mid b_i, \alpha) = \sum_{s' \in S} \sum_{s \in S,\, Poss(\alpha,s)} P(s', o \mid s, \alpha) \cdot b_i(s)$$

is the probability of observing o after executing α in b_i.

A *mixed policy* is of the form $\pi = (\pi_i)_{i \in I}$, where each π_i assigns to every belief state b and number of steps to go $h \in \{0, \ldots, H\}$ a probability function over A_i. The *expected H-step reward* to $i \in I$ under an initial belief state $b = (b_i)_{i \in I}$ and the mixed policy π, denoted $G_i(H, b, \pi)$, is defined as

$$\begin{cases} \sum_\alpha (\Pi_{j \in I} \pi_j[b, 0](\alpha_j)) \cdot \sum_{s \in S, Poss(\alpha,s)} b_i(s) R_i(s, \alpha) & \text{if } H = 0; \\ \sum_\alpha (\Pi_{j \in I} \pi_j[b, H](\alpha_j)) \cdot (\sum_{s \in S, Poss(\alpha,s)} b_i(s) R_i(s, \alpha) + \\ \quad \sum_{o \in O} P(b_i^{\alpha,o} | b_i, \alpha) \cdot G_i(H{-}1, b^{\alpha,o}, \pi)) & \text{otherwise.} \end{cases}$$

A policy π is a *Nash equilibrium* of G iff for each agent $i \in I$ and each belief state b, it holds that $G_i(H, b, \pi'_i \circ \pi_{-i}) \leqslant G_i(H, b, \pi_i \circ \pi_{-i})$ for all π'_i. We are especially interested in *partial Nash equilibria*, which are only defined for an initial belief state and all future belief states within a fixed horizon.

Algorithm. We characterize Nash equilibria of P by finite-horizon value iteration from local Nash equilibria of normal form games. We assume an arbitrary Nash selection function f for normal form games with action set $(A_i)_{i \in I}$. For every belief state $b = (b_i)_{i \in I}$ and number of steps to go $h \in \{0, \ldots, H\}$, we consider the normal form game $G[b, h] = (I, (A_i)_{i \in I}, (Q_i[b, h])_{i \in I})$, where $Q_i[b, h](\alpha)$ is defined as follows (for all actions α and agents $i \in I$):

$$\begin{cases} \sum_{s \in S,\, Poss(\alpha,s)} b_i(s) R_i(s, \alpha) & \text{if } h = 0; \\ \sum_{s \in S,\, Poss(\alpha,s)} b_i(s) R_i(s, \alpha) + \sum_{o \in O} P(b_i^{\alpha,o} | b_i, \alpha) \cdot v_f^i(G[b^{\alpha,o}, h{-}1]) & \text{otherwise.} \end{cases}$$

The next result shows that the above finite-horizon value iteration computes a Nash equilibrium for consistent probabilistic action descriptions P in $\mathcal{GC}+$.

Theorem 4.1. *Let P be a consistent probabilistic action description in $\mathcal{GC}+$, and $\pi = (\pi_i)_{i \in I}$ be defined by $\pi_i(b, h) = f_i(G[b, h])$ for all agents $i \in I$, belief states b, and number of steps to go $h \in \{0, \ldots, H\}$. Then, π is a Nash equilibrium of G, and $G_i(H, b, \pi) = v_f^i(G[b, H])$ for all $i \in I$ and b.*

The following theorem shows that every POSG can be encoded as a consistent probabilistic action description in $\mathcal{GC}+$.

Theorem 4.2. *Let $G = (I, S, (A_i)_{i \in I}, (O_i)_{i \in I}, P, (R_i)_{i \in I})$ be a POSG. Then, there exists a consistent probabilistic action description D in $\mathcal{GC}+$ that encodes G.*

Example 4.1 (Two Robots cont'd). Suppose the initial belief of robot a_1 (resp., a_2) is as in Fig. 1, right (resp., left) side. In particular, a_1 initially believes that o_1 is at $l_{1,2}$ or $l_{2,2}$, while a_2 initially believes that o_1 is at $l_{2,3}$ or $l_{2,2}$. Given the three possible

states $s_{1,2}$, $s_{2,2}$, and $s_{2,3}$ such that $s_{i,j} \models at(o_1){=}l_{i,j}$, let the probabilities be given by $b_1(s_{1,2}) = 0.2$ and $b_1(s_{2,2}) = 0.8$ for a_1, and by $b_2(s_{2,3}) = 0.4$ and $b_2(s_{2,2}) = 0.6$ for a_2.

How should the two robots act in such an initial situation? We now apply our finite-horizon value iteration algorithm to compute a partial Nash equilibrium. Notice that $pickUp(a_1, l_{2,3})$ and $pickUp(a_2, l_{1,2})$ are not executable in the initial belief states of a_2 and a_1, respectively. Hence, $pickUp$ can only be executed in $l_{2,2}$. In this case, each agent wants to get to $l_{1,2}$ first, execute $pickUp$, and cross the door. Assuming a 3-step horizon, we obtain two pure partial policies α_i, one for each agent a_i: (1) at 3 steps to go, α_i assigns the action $a = goTo(a_i, l_{2,2})$ to b_i, while any executable action b_j except for $goTo(a_j, l_{2,2})$ is assigned to b_j; (2) at 2 steps to go, a_i executes $pickUp_i(a_i, l_{2,2})$ from b_i^a, while a_j avoids $goTo(a_j, l_{2,2})$ from b_j^b; (3) at 1 step to go, a_i performs $goTo(a_i, d_1)$ in any reached belief state (both after $obs(holds)$ and $obs(notHolds)$), while a_j can execute any action. Both α_1 and α_2 represent a pure partial Nash equilibrium, where the expected 3-step reward of α_1 and α_2 for the robot pair (a_1, a_2) is $(70.4, 43.2)$ and $(52.8, 57.6)$, respectively. Another Nash equilibrium can be obtained form the previous policies by randomizing the first action selection with $\pi_1(b_1, a) = 0.55$ for $a = goTo(a_1, l_{2,2})$ ($\Sigma_\beta \pi_1(b_1, \beta) = 0.45$ with $\beta \neq a$), and $\pi_2(b_2, a) = 0.56$ for $a = goTo(a_2, l_{2,2})$ ($\Sigma_\beta \pi_2(b_2, \beta) = 0.44$ with $\beta \neq a$). Depending on the first action execution, the remaining policy is defined as in α_1 or α_2. In this case, the expected 3-step reward is $G_1(3, b_1, \pi) = 30.67$ and $G_2(3, b_2, \pi) = 25.70$.

5 Reductions and Special Cases

Computing partial Nash equilibria for a probabilistic action description P and an initial belief state requires the following computations: (i) computing the set of all states for P, (ii) deciding whether an action α is executable in a state s, (iii) computing all probabilistic transitions $P(s', o \mid s, \alpha)$, and (iv) computing Nash equilibria of normal form games. Here, (ii) can be easily done in polynomial time on P, while (iv) can be done with standard technology from game theory (see especially [19]). Finally, (i) and (iii) can be reduced to reasoning in causal theories as follows.

Reduction to Causal Theories. We first recall the main concepts of causal theories [13]. A *(causal) rule* has the form $\psi \Leftarrow \phi$ with formulas ψ and ϕ, called its *head* and *body*, respectively. A *causal theory* T is a finite set of rules. Let I be an interpretation of the variables in T. The *reduct* of T relative to I, denoted T^I, is defined as $\{\psi \mid \psi \Leftarrow \phi \in T, I \models \phi\}$. We say I is a *model* of T iff I is the unique model of T^I.

The following result shows that the tasks (i) and (iii) above can be reduced to computing the set of all models of a causal theory. It follows from the original semantics of $\mathcal{C}+$ based on causal theories [13]. In the case of *definite* causal laws, where all law heads ψ in (1) and (2) are literals, the set of all models of the corresponding causal theories can be computed using the Causal Calculator and answer set programming [13].

Proposition 5.1. *Let D be an action description.*

(a) Let T be the set of all rules $\psi \Leftarrow \phi$ such that either (i) ***caused*** *ψ* ***if*** *$\phi \in D$, or (ii) $\phi = \psi = X{=}x$ for some simple fluent $X \in \mathcal{X}$ and $x \in I(X)$. Then, an interpretation s of all fluents and rigid variables is a state of D iff it is a model of T.*

(b) Let α be an action executable in state s. Let $T_{s\cup\alpha}$ be the set of all $\psi \Leftarrow \phi$ such that either (i) ***caused*** *ψ* ***if*** *$\phi \in D$, or (ii) $s \cup \alpha \models \theta$ for some* ***caused*** *ψ* ***if*** *ϕ* ***after*** *$\theta \in D$. Then, $\Phi(s, \alpha)$ is the set of all models s' of $T_{s\cup\alpha}$ that coincide with s on all rigid variables.*

Acyclic Action Descriptions. The action description of Section 3 is *acyclic*, which allows for polynomial-time computations, as we now show. A causal theory T is *acyclic* relative to $W \subseteq V$ iff (i) every rule head is a literal, and (ii) there is a mapping κ from W to the non-negative integers such that $\kappa(X) > \kappa(Y)$ for all $X, Y \in W$ such that X (resp., Y) occurs in the head (resp., body) of some rule in T. An action description D is *acyclic* iff (i) the set of all rules $\psi \Leftarrow \phi$ with **caused** ψ **if** $\phi \in D$ is acyclic relative to all statically determined fluents and rigid variables, and (ii) for each state s and action α executable in s, it holds that $T_{s\cup\alpha}$ is acyclic relative to all fluents.

The following result shows that, in the acyclic case, every interpretation of the simple fluents produces at most one state, which is computable in polynomial time. Similarly, the $\Phi(s, \alpha)$'s contain at most one state, and are computable in polynomial time.

Theorem 5.1. *Let D be an acyclic action description. Then: (a) Every interpretation f of the set of all simple fluents can be extended to at most one state s of D. (b) Deciding whether such s exists and computing it can be done in polynomial time. (c) If s is a state and α an action executable in s, then $\Phi(s, \alpha)$ is either empty or a singleton, and it is computable in polynomial time.*

6 Summary and Outlook

We have presented the action language $GC+$ for reasoning about actions in multi-agent systems under probabilistic uncertainty and partial observability, which is an extension of the action language $\mathcal{C}+$ that is inspired by partially observable stochastic games (POSGs). We have provided a finite-horizon value iteration algorithm and shown that it characterizes finite-horizon Nash equilibria. We have also given a reduction to non-monotonic causal theories and identified the special case of acyclic action descriptions in $GC+$, where transitions are computable in polynomial time.

An interesting topic of future research is to define similar action languages for more general classes of POSGs and decentralized POMDPs.

Acknowledgments. This work has been supported by the Austrian Science Fund Project P18146-N04 and a Heisenberg Professorship of the German Research Foundation. We thank the reviewers for their constructive comments, which helped to improve our work.

References

1. F. Bacchus, J. Y. Halpern, and H. J. Levesque. Reasoning about noisy sensors and effectors in the situation calculus. *Artif. Intell.*, 111(1-2):171–208, 1999.
2. C. Baral, N. Tran, and L.-C. Tuan. Reasoning about actions in a probabilistic setting. In *Proceedings AAAI-2002*, pp. 507–512, 2002.

3. D. S. Bernstein, S. Zilberstein, and N. Immerman. The complexity of decentralized control of Markov decision processes. In *Proceedings UAI-2000*, pp. 32–37, 2000.
4. C. Boutilier. Sequential optimality and coordination in multiagent systems. In *Proceedings IJCAI-1999*, pp. 478–485, 1999.
5. C. Boutilier, R. Reiter, and B. Price. Symbolic dynamic programming for first-order MDPs. In *Proceedings IJCAI-2001*, pp. 690–700, 2001.
6. C. Boutilier, R. Reiter, M. Soutchanski, and S. Thrun. Decision-theoretic, high-level agent programming in the situation calculus. In *Proceedings AAAI-2000*, pp. 355–362, 2000.
7. T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres. A logic programming approach to knowledge-state planning, II: The DLV$^{\mathcal{K}}$ system. *Artif. Intell.*, 144(1-2):157–211, 2003.
8. T. Eiter and T. Lukasiewicz. Probabilistic reasoning about actions in nonmonotonic causal theories. In *Proceedings UAI-2003*, pp. 192–199, 2003.
9. R. Emery-Montemerlo, G. Gordon, J. Schneider, and S. Thrun. Game theoretic control for robot teams. In *Proceedings ICRA-2005*, pp. 1175–1181, 2005.
10. A. Finzi and T. Lukasiewicz. Game-theoretic reasoning about actions in nonmonotonic causal theories. Report Nr. 1843-05-04, Institut für Informationssysteme, TU Wien, 2005.
11. N. H. Gardiol and L. P. Kaelbling. Envelope-based planning in relational MDPs. In *Proceedings NIPS-2003*, 2003.
12. M. Gelfond and V. Lifschitz. Representing action and change by logic programs. *J. Logic Program.*, 17:301–322, 1993.
13. E. Giunchiglia, J. Lee, V. Lifschitz, N. McCain, and H. Turner. Nonmonotonic causal theories. *Artif. Intell.*, 153(1-2):49–104, 2004.
14. C. Guestrin, D. Koller, C. Gearhart, and N. Kanodia. Generalizing plans to new environments in relational MDPs. In *Proceedings IJCAI-2003*, pp. 1003–1010, 2003.
15. E. A. Hansen, D. S. Bernstein, and S. Zilberstein. Dynamic programming for partially observable stochastic games. In *Proceedings AAAI-2004*, pp. 709–715, 2004.
16. L. P. Kaelbling, M. L. Littman, and A. R. Cassandra. Planning and acting in partially observable stochastic domains. *Artif. Intell.*, 101(1-2):99–134, 1998.
17. M. L. Littman. Markov games as a framework for multi-agent reinforcement learning. In *Proceedings ICML-1994*, pp. 157–163, 1994.
18. N. McCain and H. Turner. Causal theories of action and change. In *Proceedings AAAI-1997*, pp. 460–465, 1997.
19. R. McKelvey and A. McLennan. Computation of equilibria in finite games. In *Handbook of Computational Economics*, pp. 87–142. Elsevier, 1996.
20. R. Nair, M. Tambe, M. Yokoo, D. V. Pynadath, and S. Marsella. Taming decentralized POMDPs: Towards efficient policy computation for multiagent settings. In *Proceedings IJCAI-2003*, pp. 705–711, 2003.
21. G. Owen. *Game Theory: Second Edition*. Academic Press, 1982.
22. D. Poole. Decision theory, the situation calculus and conditional plans. *Electronic Transactions on Artificial Intelligence*, 2(1-2):105–158, 1998.
23. M. L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. Wiley, 1994.
24. R. Reiter. *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. MIT Press, 2001.
25. J. van der Wal. *Stochastic Dynamic Programming*, volume 139 of *Mathematical Centre Tracts*. Morgan Kaufmann, 1981.
26. J. von Neumann and O. Morgenstern. *The Theory of Games and Economic Behavior*. Princeton University Press, 1947.

Some Logical Properties of Nonmonotonic Causal Theories

Marek Sergot and Robert Craven

Department of Computing, Imperial College London, SW7 2AZ, UK
{mjs, rac101}@doc.ic.ac.uk

Abstract. The formalism of nonmonotonic causal theories (Giunchiglia, Lee, Lifschitz, McCain, Turner, 2004) provides a general-purpose formalism for nonmonotonic reasoning and knowledge representation, as well as a higher level, special-purpose notation, the action language $\mathcal{C}+$, for specifying and reasoning about the effects of actions and the persistence ('inertia') of facts over time. In this paper we investigate some logical properties of these formalisms. There are two motivations. From the technical point of view, we seek to gain additional insights into the properties of the languages when viewed as a species of conditional logic. From the practical point of view, we are seeking to find conditions under which two different causal theories, or two different action descriptions in $\mathcal{C}+$, can be said to be equivalent, with the further aim of helping to decide between alternative formulations when constructing practical applications.

1 Introduction

The formalism of nonmonotonic causal theories, presented by Giunchiglia, Lee, Lifschitz, McCain and Turner [1], is a general-purpose language for knowledge representation and nonmonotonic reasoning. A *causal theory* is a set of causal rules each of which is an expression of the form

$$F \Leftarrow G$$

where F and G are formulas of an underlying propositional language and $F \Leftarrow G$ corresponds to the statement "if G, then F has a cause" (which is not the same as saying that G is a cause for F).

Associated with causal theories is the action language $\mathcal{C}+$, also presented in [1]. This may be viewed as a higher-level formalism for defining classes of causal theories in a concise and natural way, for the purposes of specifying and reasoning about the effects of actions and the persistence, or 'inertia', of facts through time, with support for indirect effects, non-deterministic actions and concurrency. The two closely-related formalisms have been used to represent standard domains from the knowledge representation literature.

In this paper, we investigate some logical properties of these formalisms. There are two motivations. The first is technical, to gain new insights into the

C. Baral et al. (Eds.): LPNMR 2005, LNAI 3662, pp. 198–210, 2005.
© Springer-Verlag Berlin Heidelberg 2005

languages when they are viewed as species of conditional logic. For example, Turner [2] presents a more general formalism called the 'logic of universal causation'. A rule $F \Leftarrow G$ of a causal theory can be expressed equivalently in this logic by the formula

$$G \rightarrow \mathbf{C}\,F$$

where $\mathbf{C}$ is a modal operator standing for 'there is a cause for' (and $\rightarrow$ is truth-functional, 'material' implication). Since $\mathbf{C}$ is a normal modal operator whose logic is at least as strong as $S5$, some logical properties of $F \Leftarrow G$ are immediately obvious. For example, we can see from $G \rightarrow \mathbf{C}\,F \vdash_{S5} G \rightarrow \mathbf{C}\,(F \vee H)$ that the logic of causal theories will exhibit the property of 'weakening of the consequent': $F \Leftarrow G$ implies (in a sense to be made more precise) $(F \vee H) \Leftarrow G$. Other properties of $F \Leftarrow G$ will be straightforwardly propositional, such as 'strengthening of the antecedent': $F \Leftarrow G$ implies $F \Leftarrow G \wedge H$. This last property is intriguing, since it is often seen as a characteristic feature of monotonic conditionals, yet the logic of causal theories is nonmonotonic.

In this paper, we will not rely on the translation to Turner's modal logic but prove properties directly from the semantics of causal theories. This is largely because of space limitations. Although many of the properties can be derived quite straightforwardly in $S5$, there is some preliminary notation and terminology we need, and we do not have space to introduce it here. Moreover, there are some fundamental properties of causal theories that are not inherited from $S5$.

The second motivation is a practical one. Causal theories and $\mathcal{C}+$ are very expressive languages. One purpose of the technical investigation is to find conditions under which which two different causal theories, or two different action descriptions in $\mathcal{C}+$, can be said to be equivalent, with the further aim of helping to decide between alternative formulations when constructing applications.

2 Causal Theories

A *multi-valued propositional signature* σ [3,1] is a set of symbols called *constants*. For each constant c in σ, there is a non-empty set $dom(c)$ of values called the *domain* of c. An *atom* of a signature σ is an expression of the form $c{=}v$, where c is a constant in σ and $v \in dom(c)$. A *formula* φ of signature σ is any propositional compound of atoms of σ.

A *Boolean constant* is one whose domain is the set of truth values $\{\mathsf{t}, \mathsf{f}\}$. If p is a Boolean constant, p is shorthand for the atom $p{=}\mathsf{t}$ and $\neg p$ for the atom $p{=}\mathsf{f}$. Notice that, as defined here, $\neg p$ is an *atom* when p is a Boolean constant.

An *interpretation* of σ is a function that maps every constant in σ to an element of its domain. An interpretation I *satisfies* an atom $c{=}v$, written $I \models c{=}v$, if $I(c) = v$. The satisfaction relation $\models$ is extended from atoms to formulas in accordance with the standard truth tables for the propositional connectives. When X is a set of formulas we also write $I \models X$ to signify that $I \models \varphi$ for all formulas $\varphi \in X$. I is then a *model* for the set of formulas X.

We write $\models_\sigma \varphi$ to mean that $I \models_\sigma \varphi$ for all interpretations I of σ. Where X is a set of formulas of signature σ, $X \models_\sigma \varphi$ denotes that $I \models_\sigma \varphi$ for all

interpretations I of σ such that $I \models_\sigma X$. When X' is a set of formulas of signature σ, $X \models_\sigma X'$ is shorthand for $X \models_\sigma \varphi$ for all formulas $\varphi \in X'$. In addition, where A and B are sets of formulas of a multi-valued propositional signature, we define $A \equiv_\sigma B$ to mean that $A \models_\sigma B$ and $B \models_\sigma A$. A *causal rule* is an expression of the form $F \Leftarrow G$, where F and G are formulas of signature σ. A *causal theory* is a set of causal rules.

Semantics. Let Γ be a causal theory, and let X be an interpretation of its underlying propositional signature. Then the *reduct* of Γ, written Γ^X, is

$$\{F \mid F \Leftarrow G \in \Gamma \text{ and } X \models G\}$$

X is a model of Γ, written $X \models_{\mathsf{C}} \Gamma$, iff X is the unique model of the reduct Γ^X. By $models(\Gamma)$ we denote the set of all models of the causal theory Γ.

Γ^X is the set of all formulas which have a cause to be true, according to the rules of Γ, under the interpretation X. If Γ^X has no models, or has more than one model, or if it has a unique model different from X, then X is not considered to be a model of Γ. Γ is *consistent* or *satisfiable* iff it has a model.

For an illustration of the preceding definitions, consider the causal theory T_1, with underlying Boolean signature $\{p, q\}$: $T_1 = \{p \Leftarrow q,\ q \Leftarrow q,\ \neg q \Leftarrow \neg q\}$.

There are clearly four possible interpretations of the signature: $X_1(p \mapsto \mathsf{t},\ q \mapsto \mathsf{t})$, $X_2(p \mapsto \mathsf{t},\ q \mapsto \mathsf{f})$, $X_3(p \mapsto \mathsf{f},\ q \mapsto \mathsf{t})$, and $X_4(p \mapsto \mathsf{f},\ q \mapsto \mathsf{f})$. It is clear that $T_1^{X_1} = \{p, q\}$, whose only model is X_1; $T_1^{X_2} = \{\neg q\}$, which has two models; $T_1^{X_3} = \{p, q\}$, whose only model is $X_1 \neq X_3$; and $T_1^{X_4} = \{\neg q\}$, which has two models. In only one of these cases—that of X_1—is it true that the reduct of the causal theory with respect to the interpretation has that interpretation as its unique model. Thus $X_1 \models_{\mathsf{C}} T_1$ and $models(T_1) = \{X_1\}$.

Suppose we add another law to T_1: for example, $T_2 = T_1 \cup \{\neg p \Leftarrow \neg p\}$. Now we have $models(T_2) = \{X_1, X_4\}$. In this example, augmenting the causal theory *increases* the set of models. It is clear that in general, for causal theories Γ and Δ, $models(\Gamma \cup \Delta) \not\subseteq models(\Gamma)$. This is the sense in which the causal theories are nonmonotonic. In the following, one of our purposes will be to invesigate under which conditions $\Gamma \cup \Delta$ has the same models as Γ.

3 A Consequence Relation Between Causal Theories

In this section, we frequently omit set-theoretic brackets from causal theories where doing so does not create confusion. In particular, causal theories which are singletons are often represented by the sole law they contain.

Proposition 1. *$X \models_{\mathsf{C}} \Gamma$ iff, for every formula F, $X \models F$ iff $\Gamma^X \models_\sigma F$.*

Proof. This is Proposition 1 of [1]. □

Observation 2. $(\Gamma_1 \cup \Gamma_2)^X = \Gamma_1^X \cup \Gamma_2^X$.

Proposition 3. *$X \models \{F \Leftarrow G\}^X$ iff $X \models G \rightarrow F$.*

Proof. Assume $X \models \{F \Leftarrow G\}^X$. If $X \models G$, then $\{F \Leftarrow G\}^X = \{F\}$, so $X \models F$. For the other direction, suppose $X \models G \rightarrow F$. If $X \models G$ then $X \models F$. But then $\{F \Leftarrow G\}^X = \{F\}$ and we have $X \models \{F \Leftarrow G\}^X$. If $X \not\models G$ then $\{F \Leftarrow G\}^X = \emptyset$, and $X \models \{F \Leftarrow G\}^X$, trivially. □

It follows from the above that $X \models \{F_1 \Leftarrow G_1, \ldots, F_n \Leftarrow G_n\}^X$ iff $X \models (G_1 \rightarrow F_1) \wedge \cdots \wedge (G_n \rightarrow F_n)$. Moreover, if a causal theory Γ contains a rule $F \Leftarrow G$ then every model of Γ satisfies $G \rightarrow F$, i.e., $X \models_{\mathsf{C}} \Gamma$ implies $X \models G \rightarrow F$. This last remark is Proposition 2 of [1].

Where Γ is a causal theory, we will denote by $mat(\Gamma)$ the set of formulas obtained by replacing every rule $F \Leftarrow G$ of Γ by the corresponding material implication, $G \rightarrow F$. The remarks above can thus be summarised as follows.

Proposition 4.

$$(i)\ X \models \Gamma^X \text{ iff } X \models mat(\Gamma) \qquad (ii)\ X \models_{\mathsf{C}} \Gamma \text{ implies } X \models mat(\Gamma)$$

Proof. In the preceding discussion. □

We now define a notion of consequence between causal theories. This will allow us to say under which conditions two causal theories are equivalent, to simplify causal theories by removing causal laws that are implied by the causal theory, and to identify (in the following section) general properties of causal laws.

We will say that causal theories Γ_1 and Γ_2 of signature σ are equivalent, written $\Gamma_1 \equiv \Gamma_2$, when $\Delta \cup \Gamma_1$ and $\Delta \cup \Gamma_2$ have the same models for all causal theories Δ of signature σ. We will say that Γ_1 implies Γ_2, written $\Gamma_1 \vdash \Gamma_2$, when $(\Gamma_1 \cup \Gamma_2) \equiv \Gamma_1$, that is, when $\Delta \cup \Gamma_1 \cup \Gamma_2$ and $\Delta \cup \Gamma_1$ have the same models for all causal theories Δ of signature σ.

Proposition 5. $\Gamma_1 \equiv \Gamma_2$ iff $\Gamma_1 \vdash \Gamma_2$ and $\Gamma_2 \vdash \Gamma_1$

Proof. A straightforward consequence of the definitions and basic set theory. □

Proposition 6. $\Gamma \vdash \Gamma_1$ *and* $\Gamma \vdash \Gamma_2$ *iff* $\Gamma \vdash (\Gamma_1 \cup \Gamma_2)$

Proof. Immediate from the definitions. □

Proposition 7. *For all causal theories Γ, Γ_1, Γ_2, Δ of signature σ we have:*

(*i*) *If $\Gamma_1 \equiv \Gamma_2$, then $(\Gamma_1 \cup \Delta) \vdash \Gamma$ iff $(\Gamma_2 \cup \Delta) \vdash \Gamma$.*
(*ii*) *If $\Gamma_1 \equiv \Gamma_2$, then $\Gamma \vdash (\Delta \cup \Gamma_1)$ iff $\Gamma \vdash (\Delta \cup \Gamma_2)$.*
(*iii*) *If $\Gamma_1 \equiv \Gamma_2$, then $(\Gamma_1 \cup \Delta) \equiv (\Gamma_2 \cup \Delta)$.*

Proof. Part (*ii*): suppose $\Gamma_1 \equiv \Gamma_2$ and $(\Gamma_1 \cup \Delta) \vdash \Gamma$. That $models(\Delta' \cup (\Gamma_2 \cup \Delta) \cup \Gamma)$ is equal to $models(\Delta' \cup (\Gamma_2 \cup \Delta))$ follows easily using basic set theory. The other parts can be proved in similar fashion. □

Proposition 8. *The relation $\vdash$ between causal theories of a given signature σ is a classical consequence relation (also known as a closure operator), that is, it satisfies the following three properties, for all causal theories Γ, Γ', and Δ:*

- ***inclusion:*** $\Gamma \vdash \Gamma$
- ***cut:*** $(\Gamma \cup \Delta) \vdash \Gamma'$ *and* $\Gamma \vdash \Delta$ *implies* $\Gamma \vdash \Gamma'$
- ***monotony:*** $\Gamma \vdash \Gamma'$ *implies* $(\Gamma \cup \Delta) \vdash \Gamma'$

Proof. 'Inclusion' is trivial. For 'monotony', suppose $\Gamma \vdash \Gamma'$. Then $(\Gamma \cup \Gamma') \equiv \Gamma$. We show $(\Gamma \cup \Delta \cup \Gamma') \equiv (\Gamma \cup \Delta)$ for any causal theory Δ. Clearly $(\Gamma \cup \Delta \cup \Gamma') \equiv ((\Gamma \cup \Gamma') \cup \Delta)$. And $((\Gamma \cup \Gamma') \cup \Delta) \equiv (\Gamma \cup \Delta)$ because $(\Gamma \cup \Gamma') \equiv \Gamma$. The proof for 'cut' is similar. □

Corollary 1.
$\Gamma_1 \vdash \Gamma_2$ *iff* $(\Gamma_1 \cup \Gamma_2) \equiv \Gamma_1$ *(As $\Gamma_1 \vdash \Gamma_1$ and $\Gamma_1 \vdash \Gamma_2$ iff $\Gamma_1 \equiv (\Gamma_1 \cup \Gamma_2)$.)*
If $\Gamma_1 \vdash \Gamma_2$ *and* $\Gamma_2 \vdash \Gamma_3$ *then* $\Gamma_1 \vdash \Gamma_3$. *(By 'monotony' and 'cut'.)*

Notice that although the formalism of causal theories is non-monotonic, in the sense that in general $models(\Gamma \cup \Delta) \not\subseteq models(\Gamma)$, the consequence relation $\vdash$ between causal theories is monotonic.

We now establish some simple *sufficient* conditions under which $\vdash$ holds.

Proposition 9. *$models(\Gamma_1) \subseteq models(\Gamma_2)$ iff, for all $X \in models(\Gamma_1)$, we have $\Gamma_1^X \equiv_\sigma \Gamma_2^X$.*

Proof. Omitted. □

Corollary 2. *$models(\Gamma_1) = models(\Gamma_2)$ iff we have $\Gamma_1^X \equiv_\sigma \Gamma_2^X$, for all $X \in models(\Gamma_1) \cup models(\Gamma_2)$.*

Proposition 10.

(*i*) $\Gamma_1 \vdash \Gamma_2$ *if* $\Gamma_1^X \models_\sigma \Gamma_2^X$, *for all* $X \models mat(\Gamma_1 \cup \Gamma_2)$.
(*ii*) $\Gamma_1 \vdash \Gamma_2$ *if* $\Gamma_1^X \models_\sigma \Gamma_2^X$, *for all* $X \models mat(\Gamma_1)$.
(*iii*) $\Gamma \vdash (G \Leftarrow F)$ *if* $\Gamma^X \models_\sigma G$, *for all* $X \models mat(\Gamma_1) \cup \{F\}$.

Proof. Part (*i*) follows from considering Proposition 9 and Corollary 2; the details of this have been omitted. Part (*ii*) is obtained from Part (*ii*) by strengthening the condition. Part (*iii*) follows from Part (*ii*): if $X \models F$ then $\{G \Leftarrow F\}^X = \{G\}$. If $X \not\models F$ then $\{G \Leftarrow F\}^X = \emptyset$, and so $\Gamma^X \models_\sigma \{G \Leftarrow F\}^X$, trivially. □

We record one further property for future reference. A causal rule of the form $F \Leftarrow F$ expresses that F holds by default. Adding $F \Leftarrow F$ to a causal theory Γ cannot eliminate models, though it can add to them.

Proposition 11. *$models(\Gamma) \subseteq models(\Gamma \cup \{F \Leftarrow F\})$*

Proof. Suppose $X \models_C \Gamma$, i.e., $X \models \Gamma^X$ and $Y \models \Gamma^X$ implies $Y = X$. We show (i) $X \models (\Gamma \cup \{F \Leftarrow F\})^X$, and (ii) if $Y \models (\Gamma \cup \{F \Leftarrow F\})^X$ then $Y = X$. For (i): if $X \models F$ then $(\Gamma \cup \{F \Leftarrow F\})^X = \Gamma^X \cup \{F\}$; we have both $X \models \Gamma^X$ and $X \models F$. If $X \not\models F$, then $(\Gamma \cup \{F \Leftarrow F\})^X = \Gamma^X$; we have $X \models \Gamma^X$. For (ii), suppose $Y \models (\Gamma \cup \{F \Leftarrow F\})^X$. Then $Y \models \Gamma^X$, and $X \models_C \Gamma$ implies $Y = X$. □

4 Properties of $\Leftarrow$

We can now prove properties about the logic of causal theories, using the preliminary results and definitions given in the previous section. We have chosen to name the results after Chellas's [4] taxonomy of rules of inference from modal logic, as this scheme is well-known and seems natural to us. We frequently omit the proofs in this section, which in nearly all cases are based straightforwardly on Proposition 10, possibly including reasoning by cases of interpretations. (Full proofs are available in a companion technical report.)

In the following, we will frequently use the notational convenience of writing $\frac{A}{B}$ instead of $A \vdash B$, where A and B are causal rules or sets of such.

Proposition 12. [RCM] If $F_1 \models_\sigma F_2$, then $F_1 \Leftarrow G \vdash F_2 \Leftarrow G$

Proof. From Proposition 10(*iii*), a sufficient condition for $F_1 \Leftarrow G \vdash F_2 \Leftarrow G$ is $\{F_1 \Leftarrow G\}^X \models_\sigma F_2$ for all $X \models_\sigma G$, which is just $F_1 \models_\sigma F_2$, which was given. □

Proposition 13. [RAug] If $G_1 \models_\sigma G_2$, then $F \Leftarrow G_2 \vdash F \Leftarrow G_1$

Proof. Similar to that for Proposition 12, and also using Proposition 10(*ii*). □

Given the preceding two propositions and Proposition 5, we have the following corollary, of which the second part will be dubbed [RCEA], again after [4].

Corollary 3.

(*i*) If $F_1 \equiv_\sigma F_2$, then $F_1 \Leftarrow G \equiv F_2 \Leftarrow G$

[RCEA] (*ii*) If $G_1 \equiv_\sigma G_2$, then $F \Leftarrow G_1 \equiv F \Leftarrow G_2$

Proposition 14.

$$\text{[RCK]} \qquad \text{If } F_1, \ldots, F_n \models_\sigma F, \text{ then } \quad \frac{F_1 \Leftarrow G, \ldots, F_n \Leftarrow G}{F \Leftarrow G} \ (n \geqslant 0)$$

The above are properties characteristic of 'normal conditional logics' [4]. We now move on to consider some distribution laws.

Proposition 15.

$$\text{[CC]} \quad \frac{F_1 \Leftarrow G, \ldots, F_n \Leftarrow G}{(F_1 \wedge \cdots \wedge F_n) \Leftarrow G} \qquad \text{[CM]} \quad \frac{(F_1 \wedge \cdots \wedge F_n) \Leftarrow G}{F_1 \Leftarrow G, \ldots, F_n \Leftarrow G}$$

$$\text{[DIL]} \quad \frac{F \Leftarrow G_1, \ldots, F \Leftarrow G_n}{F \Leftarrow (G_1 \vee \cdots \vee G_n)} \qquad \text{[cDIL]} \quad \frac{F \Leftarrow (G_1 \vee \cdots \vee G_n)}{F \Leftarrow G_1, \ldots, F \Leftarrow G_n}$$

There follows a network of interrelated properties which all express a form of monotonicity of the conditional $\Leftarrow$.

Proposition 16. [Aug] $F \Leftarrow G \;\vdash\; F \Leftarrow G \wedge H$

Proof. This is clearly a specific instance of [RAug]. For a direct proof: a sufficient condition for [Aug] is $\{F \Leftarrow G\}^X \models_\sigma F$ for all $X \models (G \wedge H)$. But if $X \models (G \wedge H)$, then the condition reduces to $F \models_\sigma F$, which holds. □

In fact, it can be shown that in the presence of the rule [RCEA], which we proved as Corollary 3(ii), the schema [Aug] is equivalent to the distribution law [cDIL]; constraints on space prevent our including the proof.

Proposition 17. [Contra] $\vdash F \Leftarrow \bot$

Proof. A sufficient condition for this is $\emptyset^X \vdash F$ for all $X \models \bot$, which holds trivially, since there is no such X. □

Proposition 18. $F \Leftarrow G \;\vdash\; \bot \Leftarrow \neg F \wedge G$

Proof. By Proposition 10(iii), a sufficient condition is that $\{F \Leftarrow G\}^X \models_\sigma \bot$, for all X with $X \models (\neg F \wedge G) \wedge (G \rightarrow F)$, which obtains: there is no such X. □

The converse of this proposition does not hold: $\bot \Leftarrow \neg F \wedge G \not\vdash F \Leftarrow G$. Now, since $G \models_\sigma \bot$ iff $G \equiv_\sigma \bot$, the schema [Contra] is equivalent to the rule: if $G \models_\sigma \bot$ then $\vdash F \Leftarrow G$. This is the case $n = 0$ for the following generalization of [RAug], easily proved by induction on n.

Proposition 19.

[RDIL] If $G \models_\sigma (G_1 \vee \cdots \vee G_n)$, then $\dfrac{F \Leftarrow G_1, \ldots, F \Leftarrow G_n}{F \Leftarrow G}$ $(n \geqslant 0)$

Proposition 20. [S] $F \Leftarrow G,\; G \Leftarrow H \;\vdash\; F \Leftarrow H$

A statement of the propogation of constraints, and a rule of Modus Ponens, are obvious instances of [S]:

Corollary 4.

[Constr] $\dfrac{F \Leftarrow G, \quad \bot \Leftarrow F}{\bot \Leftarrow G}$, [MP] $\dfrac{F \Leftarrow G, \quad G \Leftarrow \top}{F \Leftarrow \top}$

From $G \Leftarrow \top$, we get $\bot \Leftarrow \neg G \wedge \top$ using Proposition 18; an application of [RCEA] then gives us $\bot \Leftarrow \neg G$. Using [Contra] and [S] we then derive $F \Leftarrow \neg G$:

Proposition 21. $G \Leftarrow \top \;\vdash\; F \Leftarrow \neg G$

The rule describing propogation of constraints may be generalised to a weak resolution law for Horn-like rules:

Proposition 22. $F \Leftarrow G \wedge H,\ \ G \Leftarrow K\ \vdash\ F \Leftarrow H \wedge K$

The logic of causal theories does *not* contain the two equivalent rules

[I] $\vdash F \Leftarrow F$; [RI] If $G \models_\sigma F$, then $\vdash F \Leftarrow G$

To see this, use $\neg p \Leftarrow \neg p$ for F and consider the causal theory with the single rule $p \Leftarrow p$; $models(\{p \Leftarrow p\}) \neq models(\{p \Leftarrow p, \neg p \Leftarrow \neg p\})$ which means that we do not have $models(\Gamma \cup \emptyset) = models(\Gamma \cup \emptyset \cup \{\neg p \Leftarrow \neg p\})$ for all causal theories Γ, and so $\not\vdash \neg p \Leftarrow \neg p$. Although we do not have [I], it has already been shown (Proposition 11) that $models(\Gamma) \subseteq models(\Gamma \cup \{F \Leftarrow F\})$.

Using the same example, it can easily be seen that the logic of $\Leftarrow$ does not contain a contrapositive law: we have that $F \Leftarrow G \not\vdash \neg G \Leftarrow \neg F$.

Example. As one example of an application of these properties consider the following common patterns of causal rules:

$$\{\, F \Leftarrow F \wedge G \wedge \neg R,\ \neg F \Leftarrow R \,\} \qquad \text{and} \qquad \{\, F \Leftarrow F \wedge G,\ \neg F \Leftarrow R \,\}$$

In each case the first law expresses that F holds by default if G holds, and the second that R is an exception to the default rule. These pairs of laws are equivalent in causal theories. One direction is straightforward: $F \Leftarrow F \wedge G \wedge \neg R$ follows from $F \Leftarrow F \wedge G$ by [Aug].

For the other direction, notice first that $\neg F \Leftarrow R$ implies $\bot \Leftarrow F \wedge G \wedge R$ (because $\neg F \Leftarrow R$ implies $\bot \Leftarrow F \wedge R$ and the rest follows by [Aug]). Now $\vdash F \Leftarrow \bot$ [Contra], and by [S] we derive $F \Leftarrow F \wedge G \wedge R$. For the final step

$$\frac{F \Leftarrow F \wedge G \wedge R, \quad F \Leftarrow F \wedge G \wedge \neg R}{F \Leftarrow F \wedge G \wedge (R \vee \neg R)}$$

from which $F \Leftarrow F \wedge G$ follows.

5 The Action Language $\mathcal{C}+$

As with the logic of causal theories, the language $\mathcal{C}+$ is based on a multi-valued propositional signature σ, with σ partitioned into a set σ^f of *fluent constants* and a set σ^a of *action constants*. Further, the fluent constants are partitioned into those which are *simple* and those which are *statically determined*. A *fluent formula* is a formula whose constants all belong to σ^f; an action formula has at least one action constant and no fluent constants.

A *static law* is an expression of the form

$$\textsf{caused } F \textsf{ if } G,$$

where F and G are fluent formulas. An *action dynamic law* is an expression of the same form in which F is an action formula and G is a formula. A *fluent dynamic law* has the form

$$\textsf{caused } F \textsf{ if } G \textsf{ after } H,$$

where F and G are fluent formulas and H is a formula, with the restriction that F must not contain statically determined fluents. *Causal laws* are static laws or dynamic laws, and an *action description* is a set of causal laws.

In the following section we will make use of several of the many abbreviations afforded in $\mathcal{C}+$. In particular:

α causes F if G abbreviates the fluent dynamic law caused F if $\top$ after $\alpha \wedge G$;
nonexecutable α if G expresses that there is no transition of type α from a state satisfying fluent formula G. It is shorthand for the fluent dynamic law caused $\bot$ if $\top$ after $\alpha \wedge G$;
inertial f where f is a simple fluent constant, states that the values of f persist by default—they are subject to inertia—from one state to the next. It stands for the collection of fluent dynamic laws caused $f{=}v$ if $f{=}v$ after $f{=}v$ for every $v \in dom(f)$.
exogenous a where a is an action constant, stands for the set of action dynamic laws caused $a{=}v$ if $a{=}v$ after $a{=}v$ for every $v \in dom(a)$.

The language $\mathcal{C}+$ can be viewed as a useful shorthand for the logic of causal theories, for to every action description D of $\mathcal{C}+$ and non-negative integer m, there corresponds a causal theory Γ_m^D. The signature of Γ_m^D contains constants $c[i]$, such that

- $i \in \{0, \dots, m\}$ and c is a fluent constant of the signature of D, or
- $i \in \{0, \dots, m{-}1\}$ and c is an action constant of the signature of D,

and the domains of such constants $c[i]$ are kept identical to those of their constituents c. The expression $F[i]$, where F is a formula, denotes the result of suffixing $[i]$ to every occurrence of a constant in F. The causal rules of Γ_m^D are: $F[i] \Leftarrow G[i]$, for every static law in D and every $i \in \{0, \dots, m\}$, and for every action dynamic law in D and every $i \in \{0, \dots, m{-}1\}$; $F[i+1] \Leftarrow G[i+1] \wedge H[i]$, for every fluent dynamic law in D and every $i \in \{0, \dots, m{-}1\}$; and $f[0]{=}v \Leftarrow f[0]{=}v$, for every simple fluent constant f and $v \in dom(c)$.

Each action description of $\mathcal{C}+$ defines a labelled transition system. The states of the transition system are the models of Γ_0^D. Each state is an interpretation of the fluent constants σ^f. Transitions are the models of Γ_1^D. Each transition label (also called an 'event') is an interpretation of the action constants σ^a. When α is a formula of σ^a, we say that a transition label/event is of type α when it satisfies the formula α. It can also be shown [1] that paths of length m of the transition system correspond to the models of Γ_m^D.

Since action descriptions are a shorthand for particular forms of causal theories, we obtain a notion of equivalence and consequence between action descriptions of $\mathcal{C}+$: $D_1 \vdash_{\mathcal{C}+} D_2$ is defined as $\Gamma_m^{D_1} \vdash \Gamma_m^{D_2}$ for all non-negative integers m, and likewise for equivalence of action descriptions. We also inherit properties of causal laws of $\mathcal{C}+$ corresponding to those proven in Section 4. For example: α causes F if $G \vdash_{\mathcal{C}+} \alpha{\wedge}\beta$ causes F if G follows directly from 'augmentation' [Aug] for causal theories in general. In the following we will use the same labels for properties of $\mathcal{C}+$ as for the causal-theoretic properties from which they are derived.

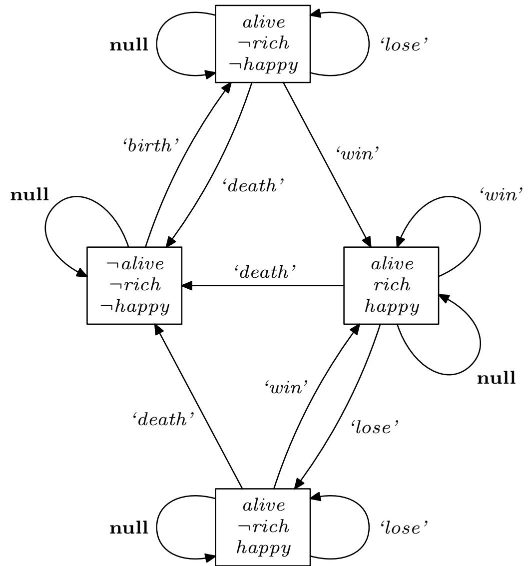
6 Example (Winning the Lottery)

Winning the lottery causes one to become (or remain) rich. Losing one's wallet causes one to become (or remain) not rich. A person who is rich is happy. A person who is not alive is neither rich nor happy.

The example is constructed partly to show how $\mathcal{C}+$ deals with indirect effects of actions (ramifications). The example also illustrates some issues in the representation of concurrent actions, actions with defeasible effects, and nondeterministic actions. Naturally it is not possible to illustrate everything with one simple example, but the example is indicative of the issues that are encountered when formulating applications in a language as expressive as $\mathcal{C}+$.

Signature: simple Boolean fluent constants $alive, rich, happy$; Boolean action constants $birth, death, win, lose$.

inertial $alive,\ rich,\ happy$
exogenous $birth,\ death,\ win,\ lose$
$birth$ causes $alive$
nonexecutable $birth$ if $alive$
$death$ causes $\neg alive$
nonexecutable $death$ if $\neg alive$
win causes $rich$
nonexecutable win if $\neg alive$
$lose$ causes $\neg rich$
nonexecutable $lose$ if $\neg alive$
caused $happy$ if $rich$
caused $\neg rich$ if $\neg alive$
caused $\neg happy$ if $\neg alive$
nonexecutable $birth \wedge death$
nonexecutable $birth \wedge win$
nonexecutable $birth \wedge lose$
nonexecutable $win \wedge lose$



States and transition labels/events are interpretations of the fluent constants and action constants, respectively. Here, each state and each transition label/event is represented by the set of atoms that it satisfies. Because of the static laws, there are only four states in the transition system and not $2^3 = 8$. The diagram label '*birth*' is shorthand for the label/event $\{birth, \neg death, \neg win, \neg lose\}$, and likewise for the labels '*death*', '*win*' and '*lose*'. The label **null** is shorthand for $\{\neg birth, \neg death, \neg win, \neg lose\}$. The diagram does not show transitions of type $death \wedge lose$, $win \wedge death$, and so on. We will discuss those presently.

Notice that *happy* is declared inertial, and so still persists even if one becomes not rich. That is why the '*lose*' transition from state $\{alive, rich, happy\}$ results in the state $\{alive, \neg rich, happy\}$. We could of course modify the action description so that *happy* is no longer inertial but defined to be true if and only

if *rich* is true. Or we might prefer to make *happy* non-inertial and let the '*lose*' transition be non-deterministic. The interactions between these various adjustments are rather subtle, however, and are not always immediately obvious.

We will restrict attention to the following two questions. First, there are alternative ways of formulating the constraints that a person cannot be rich or happy when not alive, and these alternatives have different interactions with the other causal laws. Second, as it turns out, the last group of four nonexecutable statements are all redundant, in that they are already implied by the other causal laws. There are some remaining questions about the effects of concurrent actions in the example which we will seek to identify.

First, let us look at some effects of individual actions. With the static constraints as formulated above, we have the following implied laws. (Henceforth we omit the keyword caused to conserve space.) $death$ causes $\neg alive$ (in other words, $\neg alive$ if $\top$ after $death$) together with $\neg rich$ if $\neg alive$ imply $death$ causes $\neg rich$. And in general

$$\frac{\alpha \text{ causes } F \text{ if } G, \quad F' \text{ if } F}{\alpha \text{ causes } F' \text{ if } G}$$

as is easily checked. By a similar argument we also have the implied causal law $death$ causes $\neg happy$ ($\neg happy$ if $\top$ after $death$) and win causes $happy$. We do not get the law $lose$ causes $\neg happy$ because as formulated here, we do not have the static law (explicit or implied) $\neg happy$ if $\neg rich$.

Suppose that instead of the static laws $\neg rich$ if $\neg alive$ and $\neg happy$ if $\neg alive$, we had included only the weaker constraints $\bot$ if $rich \wedge \neg alive$ and $\bot$ if $happy \wedge \neg alive$. These constraints eliminate the unwanted states, but are too weak to give the implied effects (ramifications). We also lose transitions: if $\neg happy$ if $\neg alive$ is replaced by either of $alive$ if $happy$ or $\bot$ if $happy \wedge \neg alive$, the only way that $\neg happy$ can be 'caused' is by inertia. Consequently, we eliminate all the $death$ transitions from states in which $happy$ holds: we get the implied law nonexecutable $death$ if $happy$. (We omit the formal derivation of this implied law for lack of space. It is rather involved since it also requires to taking into account the presence of other causal laws in the example.) Similarly, if we replace $\neg rich$ if $\neg alive$ by either of $alive$ if $rich$ or $\bot$ if $rich \wedge \neg alive$, the only way that $\neg rich$ can be 'caused' is by a $lose$ transition or by inertia. Consequently, transitions of type $death \wedge \neg lose$ become non-executable in the states $\{alive, rich, \neg happy\}$ and $\{alive, rich, happy\}$ whether or not we also make the earlier adjustment to the $alive/happy$ constraint. In addition, we have the implied law nonexecutable $death \wedge \neg lose$ if $rich$: a rich person cannot die unless he simultaneously loses his wallet.

There is one way in which we can use constraints $\bot$ if $rich \wedge \neg alive$ and $\bot$ if $happy \wedge \neg alive$ (or $alive$ if $rich$ and $alive$ if $happy$) without losing transitions. That is by adding a pair of extra fluent dynamic laws: either

$$death \text{ causes } \neg rich \qquad \text{and} \qquad death \text{ causes } \neg happy$$

or the weaker pair $death$ may cause $\neg rich$ and $death$ may cause $\neg happy$. (In $\mathcal{C}+$, α may cause F is an abbreviation for the fluent dynamic law F if F after α.) We leave out the (straightforward) derivation that demonstrates both these pairs

have the claimed effect. Neither is entirely satisfactory since they require all ramifications of *death* to be identified in advance and then modelled explictly using causal laws.

We turn now to examine the effects of concurrent actions. First, notice that the law nonexecutable $birth \wedge death$ is implied by the other causal laws. Because: $alive$ if $\top$ after $birth$ and $\neg alive$ if $\top$ after $death$ imply by [Aug] $alive$ if $\top$ after $birth \wedge death$ and $\neg alive$ if $\top$ after $birth \wedge death$, which in turn together imply by [CC] $alive \wedge \neg alive$ if $\top$ after $birth \wedge death$ (which is equivalent to nonexecutable $birth \wedge death$). And in general

$$\frac{\alpha \text{ causes } A \text{ if } F, \quad \beta \text{ causes } B \text{ if } G, \quad C \text{ if } A \wedge B}{\alpha \wedge \beta \text{ causes } C \text{ if } F \wedge G}$$

There is another derivation of nonexecutable $birth \wedge death$ from the causal laws of the example. We have the causal laws nonexecutable $birth$ if $alive$ and nonexecutable $death$ if $\neg alive$. $\bot$ if $\top$ after $birth \wedge alive$ and $\bot$ if $\top$ after $death \wedge \neg alive$ imply by [Aug]: $\bot$ if $\top$ after $birth \wedge death \wedge alive$ and $\bot$ if $\top$ after $birth \wedge death \wedge \neg alive$, which in turn together imply by [DIL] $\bot$ if $\top$ after $(birth \wedge death \wedge alive) \vee (birth \wedge death \wedge \neg alive)$, whose antecedent can be simplified by [RCEA]: $\bot$ if $\top$ after $birth \wedge death$.

In general we have:

$$\frac{\text{nonexecutable } \alpha \text{ if } F, \quad \text{nonexecutable } \beta \text{ if } G}{\text{nonexecutable } \alpha \wedge \beta \;\text{ if } (F \vee G)}$$

What of $birth \wedge win$ and $birth \wedge lose$? We have

$$\frac{\text{nonexecutable } birth \text{ if } alive, \quad \text{nonexecutable } win \text{ if } \neg alive}{\text{nonexecutable } birth \wedge win}$$

from which nonexecutable $birth \wedge lose$ follows by a similar argument.

This leaves transitions of type $death \wedge lose$ and $death \wedge win$. $death \wedge lose$ is not problematic. We have the implied causal laws $death \wedge lose$ causes $\neg alive$ (by [Aug] from $death$ causes $\neg alive$) and $death \wedge lose$ causes $\neg rich$ (either by [Aug] from $lose$ causes $\neg rich$ or from the implied law $death$ causes $\neg rich$). In this example, the effects of $death \wedge lose$ transitions are the same as those of $death \wedge \neg lose$ transitions.

Consider now $death \wedge win$. Here we need some adjustment to the example's formulation. We have the implied law nonexecutable $win \wedge death$ because (one of several possible derivations): we have the implied law $(win \wedge death)$ causes $(rich \wedge \neg alive)$, the static law $\neg rich$ if $\neg alive$ implies $\bot$ if $rich \wedge \neg alive$, and so $(win \wedge death)$ causes $\bot$, which is equivalent to nonexecutable $win \wedge death$.

But it seems unreasonable to insist that $win \wedge death$ transitions cannot happen—that was not the intention when the example was originally formulated. We can admit the possibility of $win \wedge death$ transitions by re-formulating the relevant causes statement for win so that it reads instead win causes $rich$ if $\neg death$, or equivalently $win \wedge \neg death$ causes $rich$. The effects of the '*win*' transitions are unchanged, but the transition system now contains transitions of type $win \wedge death$: their effects are exactly the same as those of '*death*' and $death \wedge lose$ transitions.

But note that after this adjustment, we have to re-examine other combinations of possible concurrent actions. $win \wedge birth$ is still non-executable (it depended on the pre-conditions of the two actions, not their effects) but we no longer have nonexecutable $win \wedge lose$. We have only the implied law $(win \wedge lose)$ causes $(rich \wedge \neg rich)$ if $\neg death$, or equivalently, nonexecutable $win \wedge lose \wedge \neg death$. So now a person can win the lottery and lose his wallet simultaneously, but only if he dies at the same time.

But suppose $win \wedge lose \wedge \neg death$ *is* intended to be executable. What should its effects be? One possibility is that the effects of win override those of $lose$. We replace the $lose$ causes $\neg rich$ law by the weaker $lose$ causes $\neg rich$ if $\neg win$. A second possibility is that the effects of $lose$ override those of win. We replace the win causes $rich$ if $\neg death$ law by the weaker win causes $rich$ if $\neg death \wedge \neg lose$. (And we may prefer to introduce an 'abnormality' action constant (see [1, Section 4.3]) to express the defeasibility of winning more concisely.) The third possibility is to say that $win \wedge lose$ transitions are non-deterministic:

$$win \wedge lose \text{ may cause } rich, \qquad win \wedge lose \text{ may cause } \neg rich$$

What of the interactions between non-deterministic $win \wedge lose$ actions and $death$? We still have the implied law $win \wedge lose \wedge death$ causes $\neg rich$. But perhaps the non-deterministic effects of the other $win \wedge lose$ transitions should have been formulated thus:

$$win \wedge lose \wedge \neg death \text{ may cause } rich, \qquad win \wedge lose \wedge \neg death \text{ may cause } \neg rich$$

This is unnecessary. In $\mathcal{C}+$ $\{\alpha$ may cause $F,\ \alpha$ may cause $\neg F, \beta$ causes $\neg F\}$ and $\{\alpha \wedge \neg\beta$ may cause $F,\ \alpha \wedge \neg\beta$ may cause $\neg F,\ \beta$ causes $\neg F\}$ are equivalent. Left-to-right is just an instance of [Aug]. For right-to-left, notice that $\alpha \wedge \neg\beta$ may cause F, β causes $\neg F$ is an instance of the general pattern of causal rules $\{P \Leftarrow P \wedge Q \wedge \neg R,\ \neg P \Leftarrow R\}$, discussed at the end of Section 4. It is equivalent to $\{P \Leftarrow P \wedge Q,\ \neg P \Leftarrow R\}$. For the other part, notice that β causes $\neg F$ implies $\alpha \wedge \beta$ may cause $\neg F$ by [Aug], and $\alpha \wedge \neg\beta$ may cause $\neg F$ and $\alpha \wedge \beta$ may cause $\neg F$ together imply α may cause $\neg F$ by [DIL] and [RCEA].

There are other variations of the example that we do not consider here for lack of space. We might remove the declaration that $happy$ is inertial. Or we might choose to make the fluent constant $happy$ statically determined instead of 'simple'. These changes have a further set of interactions with the other causal laws. Their effects can be analysed in similar fashion.

References

1. Giunchiglia, E., Lee, J., Lifschitz, V., McCain, N., Turner, H.: Nonmonotonic causal theories. Artificial Intelligence **153** (2004) 49–104
2. Turner, H.: A logic of universal causation. Artificial Intelligence **113** (1999) 87–123
3. Giunchiglia, E., Lee, J., Lifschitz, V., Turner, H.: Causal laws and multi-valued fluents. In: Proc. of the Fourth Workshop on Nonmonotonic Reasoning, Action, and Change, Seattle. (2001)
4. Chellas, B.F.: Modal Logic—An Introduction. Cambridge University Press (1980)

$\mathcal{M}$odular-$\mathcal{E}$: An Elaboration Tolerant Approach to the Ramification and Qualification Problems

Antonis Kakas[1], Loizos Michael[2], and Rob Miller[3]

[1] University of Cyprus, P. O. Box 20537, CY-1678, Cyprus
antonis@ucy.ac.cy
[2] Harvard University, Cambridge, MA 02138, USA
loizos@eecs.harvard.edu
[3] University College London, London WC1E 6BT, UK
rsm@ucl.ac.uk

Abstract. We describe $\mathcal{M}$odular-$\mathcal{E}$ ($\mathcal{ME}$), a specialized, model-theoretic logic for narrative reasoning about actions, able to represent non-deterministic domains involving concurrency, static laws (constraints) and indirect effects (ramifications). We give formal results which characterize $\mathcal{ME}$'s high degree of modularity and elaboration tolerance, and show how these properties help to separate out, and provide a principled solutions to, the endogenous and exogenous qualification problems. We also show how a notion of (micro) processes can be used to facilitate reasoning at the dual levels of temporal granularity necessary for narrative-based domains involving "instantaneous" series of indirect and knock-on effects.

1 Introduction

Domain descriptions for reasoning about actions and change (RAC) in commonsense reasoning and other contexts should be *Elaboration Tolerant* [10,9]. Formalisms should be able to incorporate new information gracefully into representations, e.g. by the simple addition of sentences. Elaboration Tolerance (ET) is strongly linked with the need to have a *modular* semantics for RAC frameworks that properly separates different aspects of the domain knowledge, as argued e.g. in [6]. ET and modularity are known to be strongly related to the *Qualification Problem* in RAC – if the effect laws (or action executability laws) of our domain are not qualified in a complete way they can lead to unintended conclusions that contradict new information. In particular, new narrative information about observations or attempted actions can render the domain description inconsistent in this way.

In this paper, we present the language $\mathcal{M}$odular-$\mathcal{E}$ ($\mathcal{ME}$) as a case study in developing modular semantics for RAC frameworks in order to provide comprehensive solutions to the ramification and qualification problems. Our approach builds upon [7] and is inspired by [13], separating out the qualification problem into two parts - an *endogenous* aspect concerning qualifications expressible in the known domain language, and an *exogenous* aspect where change is qualified by

C. Baral et al. (Eds.): LPNMR 2005, LNAI 3662, pp. 211–226, 2005.
© Springer-Verlag Berlin Heidelberg 2005

unrepresented (or exogenous) factors. The semantics of $\mathcal{ME}$ decouples these two problems, allowing exogenous qualifications to come into play only when the endogenous qualification alone is not sufficient to avoid inconsistency. It uses a simple default minimization of exogenous qualifications to "minimize *unexplained* failure" (c.f. [13]) when observations of properties cannot be reconciled with the assumed success of the applied effect laws. $\mathcal{ME}$'s modular semantics offers a clean solution to the problem of anomalous models that arose from earlier treatments of the qualification problem.

To achieve the semantic decoupling of endogenous and exogenous qualifications it is important to address two issues. First, a proper treatment of ramifications, including non-determinism and loops in chains of instantaneous effects, is needed (as any incomplete treatment will cause some endogenous qualifications to be treated as exogenous). $\mathcal{ME}$ uses a notion of *processes* for this. Second, for the same reason a full account is needed for the qualifications that static constraints provide for causal laws. In this regard we distinguish between *local or explicit* and *global or implicit* qualification. Local qualifications are the explicit preconditions included in individual causal effect laws and action executability statements. Global qualifications are formed at the semantic level by taking into account static laws and interactions between effect laws. Global qualification is closely related to modularity. Without it elaboration tolerance is compromised by the need to manually reconcile each local set of qualifications with each new static law.

We show that this analysis of the qualification and ramification problems indeed results in modularity and elaboration tolerance. For example, $\mathcal{ME}$ enjoys a "free will" property – a domain description can be extended with any action attempt at any time after its recorded observations without affecting the conclusions about the domain up to that time.

2 $\mathcal{ME}$ Syntax and Examples

In this section we give $\mathcal{ME}$'s syntax and sketch its important characteristics via a series of examples.

Definition 1 (Domain Language). *An $\mathcal{ME}$* ***domain language*** *is a tuple $\langle \Pi, \preceq, \Delta, \Phi \rangle$, where $\preceq$ is a total ordering defined over the non-empty set Π of time-points, Δ is a non-empty set of action constants, and Φ is a non-empty set of fluent constants.*

Definition 2 (Formula, Literal and Conjunction). *A* ***fluent formula*** *is a propositional formula containing only fluent constants (used as extra-logical symbols), the standard connectives $\neg$, $\rightarrow$, $\leftarrow$, $\leftrightarrow$, $\vee$ and $\wedge$, and the truth value constants $\top$ and $\bot$. A* ***fluent literal*** *is either a fluent constant or its negation. An* ***action literal*** *is either an action constant or its negation. A* ***fluent conjunction*** *is a conjunction of fluent literals.*

Definition 3 (Converse). *Let E be an action or fluent constant. The* ***converse*** *of E, written $\overline{E}$, is $\neg E$, and the converse of $\neg E$, written $\overline{\neg E}$, is E.*

Definition 4 (Domain Description or Theory). *A **domain description or theory** in $\mathcal{ME}$ is a collection of the following types of statements, where ϕ is a fluent formula, T is a time point (assume an integer or real number unless otherwise stated), A is an action constant, C is a (possibly empty) set of fluent and action literals, L is a fluent literal, and E is a non-empty set of action constants and fluent literals:*

- ***h-propositions*** *of the form:* ϕ `holds-at` T
- ***o-propositions*** *of the form:* A `occurs-at` T
- ***c-propositions*** *of the form:* C `causes` L
- ***p-propositions*** *of the form:* ϕ `prevents` E
- ***a-propositions*** *of the form:* `always` ϕ

*A domain description is **finite** if it contains only a finite number of propositions.*

Singleton sets of fluent or action literals in c-propositions of the form $\{P\}$ will sometimes be written without enclosing braces, i.e. as P.

The intended meaning of h-propositions is straightforward – they can be used to record "observations" about the domain along the time line. "A `occurs-at` T" means that an attempt to execute A occurs at T. Together, the h- and o-propositions describe the "narrative" component of a domain description. "C `causes` L" means that, at any time-point, the combination of actions, inactions and preconditions described via C will *provisionally* cause L to hold immediately afterwards. As we shall see, the provisos automatically accompanying this causal rule are crucial – in any model the potential effect L competes with other potential effects, and maybe overridden, for example, because it would otherwise result in a more-than-instantaneous violation of a domain constraint described with an a-proposition. The rule "C `causes` L" is thus qualified both locally (via C) and globally via the total set of c-, p- and a-propositions. "ϕ `prevents` E" means that the circumstances described by ϕ prevent the simultaneous causation/execution of the effects/actions listed in E. " `always` ϕ" means that $\neg\phi$ can never hold, other than in temporary, instantaneous "transition states" which form part of an instantaneous chain of indirect effects. In other words, " `always` ϕ" describes a domain constraint or static law at the granularity of observable time.

Example 1 (Lift Door). A lift door can be opened and closed by pressing the "open" and "close" buttons respectively. The door is initially open, and both buttons are pressed simultaneously. This scenario can be described with a single fluent *DoorOpen* and two actions *PressOpen* and *PressClose*:

$\{PressOpen\}$ `causes` $DoorOpen$ (LD1)
$\{PressClose\}$ `causes` $\neg DoorOpen$ (LD2)
$DoorOpen$ `holds-at` 1 (LD3)
$PressOpen$ `occurs-at` 2 (LD4)
$PressClose$ `occurs-at` 2 (LD5)

Example 1 results in two models – one in which the door is open at times after 2 and one in which the door is closed. Note that, even though the conflicting

actions are not prevented from occurring together (i.e. there is no p-proposition "$\top$ `prevents` $\{PressOpen, PressClose\}$"), they do not give rise to inconsistency. More generally, we show in Section 4 that $\mathcal{ME}$ exhibits a "free will" property – from any consistent initial state, and for any given collection of c- and p-propositions, any series of actions may be attempted without giving rise to inconsistency. Put another way, any finite collection of o-, c- and p-propositions is consistent with any internally consistent collection of a-propositions. Consequently, the only way to engineer an inconsistent $\mathcal{ME}$ domain description (other than by inclusion of inconsistent a-propositions) is to include "observations" (h-propositions) along the time line which contradict the predictions that would otherwise be given by $\mathcal{ME}$'s semantics. In Section 5 we show how this remaining type of inconsistency can sometimes be overcome by attributing it to unknown exogenous reasons and applying a simple minimization to these.

The following series of "broken car" examples is to illustrate the modularity and elaboration tolerance of $\mathcal{ME}$, and how this is linked to the way a- and c-propositions interact.

Example 2 (Broken Car A). Turning the key of a car causes its engine to start running. The key is turned at time 1:

$$\{TurnKey\} \texttt{ causes } Running \qquad \text{(BC1)}$$

$$TurnKey \texttt{ occurs-at } 1 \qquad \text{(BC2)}$$

In all models of this domain the car engine is running at all times after 1. (A more complete description would typically include some local qualifications for (BC1), e.g. "$\{TurnKey, BatteryOK\}$ `causes` $Running$" – turning the key starts the engine only when the battery is OK, in which case models would also arise where e.g. $\neg BatteryOK$ and $\neg Running$ at all time-points.)

Example 3 (Broken Car B). We elaborate the previous description by stating that broken cars' engines cannot run:

$$\texttt{always } \neg(Broken \wedge Running) \qquad \text{(BC3)}$$

There are two classes of models for the elaborated domain (BC1)-(BC3) – one in which the car is broken and not running at times after 1, and one in which the car is not broken and running. The occurrence of *TurnKey* at 1 does not eliminate the model in which the car is broken because the semantics of $\mathcal{ME}$ allows (BC3) to act as a global qualification, in particular for (BC1). The *TurnKey* action does not force $\neg Broken$ at earlier times, and thus if in addition the car is known to be broken the theory remains consistent after this elaboration. Without this characteristic, we would have to alter (BC1) to "$\{TurnKey, \neg Broken\}$ `causes` $Running$" to accommodate (BC3), in other words explicitly encode as a local qualification the global qualification effect of (BC3) on (BC1). In $\mathcal{ME}$ this local qualification is redundant thus illustrating its modular nature; the a-proposition (BC3) has been simply added without further ado.

Example 4 (Broken Car C). We elaborate Example 3 with two more causal rules and an extra action occurrence:

$\{Break\}$ `causes` *Broken* (BC4)
$\{Broken\}$ `causes` $\neg$*Running* (BC5)
Break `occurs-at` 1 (BC6)

In all models of the domain (BC1)-(BC6), the car is broken and not running at times after 1. (BC5) describes an "indirect effect" or "ramification". It introduces an asymmetry between the *Running* and *Broken* fluents and their relationship with (BC3), preventing (BC3) from acting as a qualification for (BC4) in the same way as it does for (BC1). Translating global to local/explicit qualifications is therefore complex, as it requires consideration of the interactions between a- and c-propositions. $\mathcal{ME}$ deals with indirect effects by considering chains of instantaneous, temporary transition states ("nodes"). Within these causal chains, "processes" are introduced to describe the initiation and termination of fluents. These processes may "stretch" across several links of a given chain before they are complete, thus allowing all possible micro-orderings of effects to be considered. Because of the coarseness of the domain description with respect to the granularity of time, this is important for a proper treatment of collections of instantaneous effects which compete or "race" against each other. Furthermore, since the granularity of time in which these chains operate is finer than that of observable time, intermediate states within them may (temporarily) violate the static laws described by a-propositions. In Example 4, one of the chains allowed by the semantics completes the process initiating *Running* and then the process initiating *Broken*. At this point there is a state in which (BC3) is violated, but (BC5) then generates a new process terminating *Running* whose completion results in a consistent state further along the chain.

Example 5 (Broken Car B+/C+). We elaborate the previous two descriptions by observing the car running at time 2:

Running `holds-at` 2 (BC-obs)

Adding (BC-obs) to Example 3 does not result in inconsistency, but allows us to infer that the car is not broken (in particular at earlier times). Note that $\mathcal{ME}$ would facilitate the opposite conclusion (*Broken*) in exactly the same way had the observation been "$\neg$*Running* `holds-at` 2". This is because it accords exactly the same status to globally derived qualifications (in this case from (BC3)) as to qualifications localized to particular c-propositions. However, adding (BC-obs) to Example 4 does give rise to inconsistency at the level of the $\mathcal{ME}$'s "base semantics" (as detailed in Section 3), because since there are no (local or globally derived) qualifications to (BC4) and (BC5), the theory would otherwise entail $\neg$*Running*. An intuitive explanation for (BC-obs) in this context is that one or both of the effects of (BC4) and (BC5) "failed" due to exogenous circumstances (i.e. factors not included in the representation) implicitly qualifying these causal rules. This type of reasoning is captured within $\mathcal{ME}$ by the use of simple default minimization of such exogenous qualifications

(see Section 5). The minimization policy is straightforward and robust because the base semantics fully accounts for all endogenous qualifications (i.e. those expressed in the domain) by its modularity and its encapsulation of global as well as local qualifications, as described above.

Example 6 (Broken Car D). We elaborate Example 4 with the knowledge that the car was parked at time 0 in anti-theft mode (ATM), so that causing the engine to run (even for an instant) will trigger the alarm:

$$(\neg Broken \wedge \neg Running \wedge \neg Alarm \wedge ATM) \texttt{ holds-at } 0 \qquad \text{(BC7)}$$
$$\{Running, ATM\} \texttt{ causes } Alarm \qquad \text{(BC8)}$$

Intuitively, even though at times after 1 the car will be broken and not running, the alarm may or may not be triggered in this narrative, depending on whether the (indirect) effect of the *Break* action takes effect just before or just after the effect of the *TurnKey* action. This is an example of a "race" condition between competing instantaneous effects. $\mathcal{ME}$ is able to deal correctly with such representations via its processed-based semantics. It gives two models of this domain – in both models $(Broken \wedge \neg Running)$ is true at times after 1, but in one model *Alarm* is true and in the other it is false. The example illustrates how $\mathcal{ME}$'s processes operate at a finer level of temporal granularity than "observable time" in order to deal with "instantaneous" indirect effects.[1]

Example 7 (Oscillator).

$$\{On\} \texttt{ causes } \neg On \qquad \text{(OSC1)}$$
$$\{\neg On\} \texttt{ causes } On \qquad \text{(OSC2)}$$

This example (which might e.g. represent the internal mechanism of an electric buzzer) has an infinite number of models in which the truth value of *On* is arbitrarily assigned at each time point. It illustrates that $\mathcal{ME}$ is able to deal with "loops" of indirect effects without over-constraining models. It is important, for example, not to restrict the set of models to those in which the truth value of *On* alternates at each successive time-point. This is because the change within the domain is happening "instantaneously" – i.e. at an altogether finer granularity of time than "observable" time. Therefore the observable time-points are best considered as arbitrarily spaced "snapshots" of the finer-grained time continuum. A full treatment of such loops along these lines (as well as a full treatment of concurrency and nondeterminism) is necessary for $\mathcal{ME}$ to exhibit the "free will" property and resulting modularity and elaboration tolerance described above.

[1] An interesting (and more contentious) variation of Example 6 is to delete (BC4) and (BC6), and replace (BC7) with "$(Broken \wedge \neg Running \wedge \neg Alarm \wedge ATM)$ `holds-at` 0". (so that the car is already broken at 1). $\mathcal{ME}$'s semantics still gives the two models with *Alarm* true in one and false in the other. This is because it treats (BC3) only as a "stability" constraint at the temporal granularity of "observable" time, and not as a "definitional" constraint that would transcend all levels of temporal granularity. Note, however, that we could eliminate the model in which *Alarm* was true by adding the p-proposition "*Broken* `prevents` *Running*", meaning that *Broken* prevents *Running* from being caused (even instantaneously).

3 $\mathcal{M}$odular-$\mathcal{E}$ Base Semantics

In this section we give a formal account of $\mathcal{ME}$'s semantics. We begin with some straightforward preliminary definitions concerning states and processes.

3.1 Definitions Regarding States, Processes and Causal Change

Definition 5 (States and Satisfaction). *A* ***state*** *is a set S of fluent literals such that for each fluent constant F, either $F \in S$ or $\neg F \in S$ but not both. A formula ϕ* ***is satisfied in a state*** *S iff the interpretation corresponding to S is a model of ϕ.*

Definition 6 (A-Consistency). *Let D be a domain description and S a state. S is* ***a-consistent with respect to*** *D iff for every a-proposition "*`always` *ϕ" in D, ϕ is satisfied in S. D is* ***a-consistent*** *iff there exists a state which is a-consistent with respect to D. Let D_a denote the set of all a-propositions in D. Then given a fluent formula ψ, $D_a \models_a \psi$ iff ψ is entailed classically by the theory $T = \{\phi \mid$* `always` *$\phi \in D\}$.*

Definition 7 (Process). *A* ***process*** *is an expression of the form $\uparrow F$ or $\downarrow F$, where F is a fluent constant of the language. $\uparrow F$ is called* ***the initiating process of*** *F and $\downarrow F$ is called* ***the terminating process of*** *F. The* ***associated processes*** *of the c-propositions "C* `causes` *F" and "C* `causes` *$\neg F$" are respectively $\uparrow F$ and $\downarrow F$. $\uparrow F$ and $\downarrow F$ will also sometimes be written as $proc(F)$ and $proc(\neg F)$ respectively. An* ***active process log*** *is a set of processes.*

Definitions 8 – 15 concern the identification of fluent changes following instantaneously from a given state and set of actions. A *causal chain* represents a possible instantaneous series of knock-on effects implied by the causal laws. There is a repeated two-phase mechanism for constructing the "nodes" of causal chains – a triggering phase in which new processes are generated from c-propositions applicable at that point, immediately followed by a resolution phase in which some of the already-active processes complete, resulting in an update of the corresponding fluents' truth values. The process triggering is appropriately limited by the p-propositions. The triggering and completion of a particular process may be separated by several steps in the chain, so that consideration of all such chains gives an adequate treatment of "race" conditions between competing instantaneous effects. Chains terminate either because they reach a state from which no change is possible (a *static node*) or because they loop back on themselves. We have made the working (but retractable) assumption that actions trigger processes only at the beginning of such chains, at which point they are "consumed".

Definition 8 (Causal Node). *A* ***causal node*** *(or* ***node****) is a tuple $\langle S, B, P\rangle$, where S is a state, B is a set of action constants and P is an active process log. $\langle S, B, P\rangle$ is* ***fully resolved*** *iff $P = \emptyset$, and is* ***a-consistent w.r.t. a domain description*** *D iff S is a-consistent w.r.t. D.*

Definition 9 (Triggering). *Let D be a domain description, $N = \langle S, B, P \rangle$ a node, L_t a set of fluent literals, $P_t = \{proc(L) \mid L \in L_t\}$, and B_t a set of action constants. The set $(B_t \cup P_t)$ is **triggered at** N **with respect to** D iff*

1. $B_t \subseteq B$
2. *For each p-proposition "ϕ `prevents` E" in D, either ϕ is not satisfied in S or $E \not\subseteq (B_t \cup L_t)$.*
3. *For each $L \in L_t$ there is a c-proposition "C `causes` L" in D such that (i) for each action constant $A \in C$, $A \in B_t$, (ii) for each action literal $\neg A \in C$, $A \notin B_t$, and (iii) for each fluent literal $L' \in C$, $L' \in S$.*

*$(B_t \cup P_t)$ is **maximally triggered at** N **with respect to** D iff there is no other set $(B_t' \cup P_t')$ also triggered at N with respect to D and $(B_t \cup P_t)$ is a strict subset of $(B_t' \cup P_t')$.*

Definition 10 (Process Successor). *Let D be a domain description and $N = \langle S, B, P \rangle$ a node. A **process successor of** N **w.r.t.** D is a node of the form $\langle S, B_t, (P \cup P_t) \rangle$, where $(B_t \cup P_t)$ is maximally triggered at N with respect to D.*

Definition 11 (Resolvant). *Let $N = \langle S, B, P \rangle$ and $N' = \langle S', \emptyset, P' \rangle$ be causal nodes. N' is a **resolvant of** N iff $S' = S$ and $P = P' = \emptyset$ or there exists a non-empty subset R of P such that the following conditions hold.*

1. $P' = P - R$.
2. *For each fluent constant F such that both $\uparrow F$ and $\downarrow F$ are in P, either both or neither $\uparrow F$ and $\downarrow F$ are in R.*
3. *For each fluent constant F (i) if $\uparrow F \in R$ and $\downarrow F \notin R$ then $F \in S'$, (ii) if $\downarrow F \in R$ and $\uparrow F \notin R$ then $\neg F \in S'$, (iii) if $\downarrow F \notin R$ and $\uparrow F \notin R$ then $F \in S'$ iff $F \in S$.*

*N' is a **full resolvant of** N iff $P' = \emptyset$.*

Definition 12 (Stationary/Static Nodes). *Let D be a domain description and $N = \langle S, B, P \rangle$ a causal node. N is **stationary** iff for each resolvant $\langle S', \emptyset, P' \rangle$ of N, $S' = S$. N is **static w.r.t.** D iff every process successor of N w.r.t. D is stationary.*

The central definition of causal chains now follows. It is slightly complicated by the need to deal with loops – conditions 2, 3 and 4 below ensure that all chains will end when the first static or repeated node is encountered.

Definition 13 (Causal Chain). *Let D be a domain description and let N_0 be a node. A **causal chain rooted at** N_0 **with respect to** D is a (finite) sequence $N_0, N_1, ..., N_{2n}$ of nodes such that for each k, $0 \leq k \leq n-1$, N_{2k+1} is a process successor of N_{2k} w.r.t. D and N_{2k+2} is a resolvant of N_{2k+1}, and such that the following conditions hold:*

1. *N_{2n} is fully resolved.*
2. *N_{2n} is static, or there exists $k < n$ s.t. $N_{2n} = N_{2k}$.*

3. If there exists $j < k \leq n$ s.t. $N_{2j} = N_{2k}$ then $k = n$.
4. There does not exist a $k < n$ s.t. N_{2k} is static.

In the context of Example 1, Figure 1 below shows the tree of all possible causal chains with the starting node $\langle\{DoorOpen\}, \{PressClose, PressOpen\}, \emptyset\rangle$ (which intuitively corresponds to the situation at time 2). N_1 is the unique process successor of N_0, and the nodes N_2 and N_2' (which are both static) are the only resolvants of N_1.

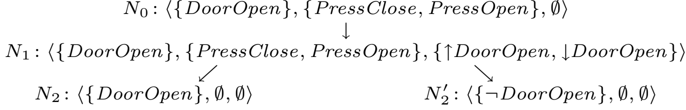


Fig. 1.

As regards Example 6, we may form several causal chains starting from the node corresponding to time 1. Here is a chain terminating with a state in which *Alarm* holds ($Br = Broken$, $Ru = Running$, $Al = Alarm$):

$N_0 : \langle\{\neg Br, \neg Ru, \neg Al, ATM\}, \{Break, TurnKey\}, \emptyset\rangle$
$N_1 : \langle\{\neg Br, \neg Ru, \neg Al, ATM\}, \{Break, TurnKey\}, \{\uparrow Br, \uparrow Ru\}\rangle$
$N_2 : \langle\{Br, Ru, \neg Al, ATM\}, \emptyset, \emptyset\rangle$
$N_3 : \langle\{Br, Ru, \neg Al, ATM\}, \emptyset, \{\downarrow Ru, \uparrow Al\}\rangle$
$N_4 : \langle\{Br, \neg Ru, Al, ATM\}, \emptyset, \emptyset\rangle$

Here is another chain terminating with a state in which $\neg Alarm$ holds:

$N_0 : \langle\{\neg Br, \neg Ru, \neg Al, ATM\}, \{Break, TurnKey\}, \emptyset\rangle$
$N_1 : \langle\{\neg Br, \neg Ru, \neg Al, ATM\}, \{Break, TurnKey\}, \{\uparrow Br, \uparrow Ru\}\rangle$
$N_2' : \langle\{Br, \neg Ru, \neg Al, ATM\}, \emptyset, \{\uparrow Ru\}\rangle$
$N_3' : \langle\{Br, \neg Ru, \neg Al, ATM\}, \emptyset, \{\downarrow Ru, \uparrow Ru\}\rangle$
$N_4' : \langle\{Br, \neg Ru, \neg Al, ATM\}, \emptyset, \emptyset\rangle$

Nodes, and in particular nodes that terminate causal chains, do not necessarily contain a-consistent states. But causal chains that do not terminate a-consistently are not discarded when computing direct and indirect instantaneous effects. Rather, the semantics identifies *proper causal descendants* within a tree of all possible causal chains starting from a given root node. These are a-consistent nodes which are either within the terminating loop of a chain (condition 1 in Definition 14), or are such that there are no other a-consistent nodes further from the root of the tree (condition 2). (For example, in Fig. 1, N_2 and N_2' are proper causal descendants of N_0 by condition 1 below, with $j = k = n = 1$.)

Definition 14 (Proper Causal Descendant). *Let D be a domain description and let N_0 and N be nodes. N is a* ***proper causal descendant of*** *N_0* ***w.r.t.***

D iff N is a-consistent w.r.t. D, and there exists a causal chain $N_0, N_1, ..., N_{2n}$ *w.r.t. D such that* $N = N_{2k}$ *for some* $0 \leq k \leq n$ *and at least one of the following two conditions holds:*

1. *There exists* $j \leq k$ *such that* $N_{2j} = N_{2n}$.
2. *There does not exist a causal chain* $N_0, N_1, ..., N_{2k}, N'_{2k+1}, ..., N'_{2m}$ *w.r.t. D and a j such that* $k < j \leq m$ *and* N'_{2j} *is a-consistent w.r.t. D.*

It is also useful to define a *stable state* as a state that does not always immediately cause its own termination (note that stable states can be in loops, but must be a-consistent):

Definition 15 (Stable State). *Let D be a domain description and let S be a state. S is **stable w.r.t.** D if there exists a node* $\langle S, \emptyset, P \rangle$ *which is a proper causal descendant of* $\langle S, \emptyset, \emptyset \rangle$.

Example 8 (Promotion). An employee gets promoted at time 1. Promotion results in a large office (LO) and big salary (BS). But nobody gets a large office when the building is overcrowded (OC), which it is at time 1:

$\texttt{always}\ \neg(OC \wedge LO)$ (PR1)

$Promote\ \texttt{causes}\ \{BS, LO\}$ (PR2)

$Promote\ \texttt{occurs-at}\ 1$ (PR3)

$(\neg LO \wedge \neg BS \wedge OC)\ \texttt{holds-at}\ 1$ (PR4)

Here is the tree of possible causal chains that arise at time 1 in this example, with the single proper causal descendant of the root node underlined:

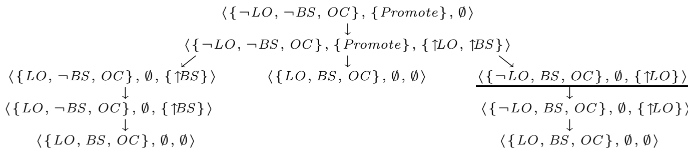


Fig. 2.

3.2 Definitions Regarding Time and Temporal Change

If a causal node corresponds to a particular time-point in the narrative of a given domain description (e.g. in Fig. 1, N_0 corresponds to time 2), then Definitions 16 – 21 below ensure that the states within its proper causal descendants indicate possible choices as to which fluents will change values in the time period immediately afterwards. These definitions are largely modifications of those in [7], but with the notion of a *change set* replacing that of initiation/termination points.

Definition 16 (Interpretation). *An **interpretation** of* $\mathcal{ME}$ *is a mapping* $H : \Phi \times \Pi \mapsto \{true, false\}$.

Definition 17 (Time-point Satisfaction). *Given a fluent formula ϕ of $\mathcal{ME}$ and a time point T, an interpretation H **satisfies** ϕ **at** T iff the mapping M_T defined by $\forall F, M_T(F) = H(F,T)$ is a model of ϕ. Given a set Z of fluent formulae, H **satisfies** Z **at** T iff H satisfies ϕ at T for each $\phi \in Z$.*

Definition 18 (State/Event Base at a Time-point). *Let D be a domain description, H an interpretation, and T a time-point. The **state at** T **w.r.t.** H, denoted $S(H,T)$, is the state $\{F \mid H(F,T) = true\} \cup \{\neg F \mid H(F,T) = false\}$. The **event base at** T **w.r.t.** D, denoted $B(D,T)$, is the set $\{A \mid$ "A `occurs-at` T" $\in D\}$.*

Definition 19 (Causal Frontier). *Let D be a domain description, T a time-point, H an interpretation and S a state. S is a **causal frontier of** H **at** T **w.r.t.** D iff there exists a node $N = \langle S, B, P\rangle$ such that N is a proper causal descendant of $\langle S(H,T), B(D,T), \emptyset\rangle$ w.r.t. D.*

Definition 20 (Change Set). *Let D be a domain description, H an interpretation, T a time-point and $\mathcal{C}$ a set of fluent literals. $\mathcal{C}$ is a **change set at** T **w.r.t.** H iff there exists a causal frontier S of H at T w.r.t. D such that $\mathcal{C} = S - S(H,T)$.*

Definition 21 (Model). *Let D be a domain description, and let Φ^* be the set of all (+ve and -ve) fluent literals in the language. Then an interpretation H is a **model** of D iff there exists a mapping $c : \Pi \mapsto 2^{\Phi^*}$ such that for all T, $c(T)$ is a change set at T w.r.t. H, and the following three conditions hold. For every fluent literal L and time-points $T_1 \prec T_3$:*

1. *If H satisfies L at T_1, and there is no time-point T_2 s.t. $T_1 \preceq T_2 \prec T_3$ and $\overline{L} \in c(T_2)$, then H satisfies L at T_3.*
2. *If $L \in c(T_1)$, and there is no time-point T_2 s.t. $T_1 \prec T_2 \prec T_3$ and $\overline{L} \in c(T_2)$, then H satisfies L at T_3.*
3. *H satisfies the following constraints:*
   - *For all "ϕ `holds-at` T" in D, H satisfies ϕ at T.*
   - *For all time-points T, $S(H,T)$ is a stable state.*

Intuitively, condition (1) above states that fluents change their truth values only via successful effects of c-propositions, and (2) states that successfully initiating a literal establishes its truth value as true. Note also that condition (3)'s requirement of stability ensures that $S(H,T)$ is a-consistent.

Definition 22 (Consistency and Entailment). *A domain description D is **consistent** if it has a model. D **entails** the h-proposition "ϕ `holds-at` T", written $D \models \phi$ `holds-at` T, iff for every model M of D, M satisfies ϕ at T.*

Example 9 (Faulty Circuit). An electric current in a faulty circuit is switched on causing a broken fuse, which in turn terminates the current:

$\{SwitchOn\}$ `causes` $ElectricCurrent$ (FC1)
$\{ElectricCurrent\}$ `causes` $BrokenFuse$ (FC2)

$\{BrokenFuse\}$ `causes` $\neg ElectricCurrent$ (FC3)
`always` $\neg(ElectricCurrent \wedge BrokenFuse)$ (FC4)
$SwitchOn$ `occurs-at` 1 (FC5)

One causal chain that could be triggered at time 1 (with non-a-consistent nodes N_4 and N_5) is:

$N_0 : \langle\{\neg ElectricCurrent, \neg BrokenFuse\}, \{SwitchOn\}, \emptyset\rangle$
$N_1 : \langle\{\neg ElectricCurrent, \neg BrokenFuse\}, \{SwitchOn\}, \{\uparrow ElectricCurrent\}\rangle$
$N_2 : \langle\{ElectricCurrent, \neg BrokenFuse\}, \emptyset, \emptyset\rangle$
$N_3 : \langle\{ElectricCurrent, \neg BrokenFuse\}, \emptyset, \{\uparrow BrokenFuse\}\rangle,$
$N_4 : \langle\{ElectricCurrent, BrokenFuse\}, \emptyset, \emptyset\rangle$
$N_5 : \langle\{ElectricCurrent, BrokenFuse\}, \emptyset, \{\downarrow ElectricCurrent\}\rangle$
$N_6 : \langle\{\neg ElectricCurrent, BrokenFuse\}, \emptyset, \emptyset\rangle.$

This chain is well-formed because N_6 is the first static resolvant node and is fully resolved (Definition 13). N_6 is a-consistent and therefore is a proper causal descendant of N_0 (Definition 14). So $\{\neg ElectricCurrent, BrokenFuse\}$ is a causal frontier at 1 of any interpretation that satisfies $(\neg ElectricCurrent \wedge \neg BrokenFuse)$ at 1 (Definition 19), thus providing the change set $\{BrokenFuse\}$ (Definition 20). Note that at the granularity level of the representation of this example, *ElectricCurrent*, the cause of *BrokenFuse*, is never true! *ElectricCurrent* is true only at a finer granularity.

4 Some Formal Results and Properties

As we have seen, $\mathcal{ME}$ provides principled, general mechanisms for causal laws to be qualified both by each other and by static laws, thus integrating all endogenous qualifications within one base-level semantic framework. $\mathcal{ME}$ also provides a high degree of modularity by its separation of information about causality (c-, p- and a-propositions), narrative information about attempted actions (o-propositions), and observations (h-propositions) within the narrative. These qualities make $\mathcal{ME}$ domain descriptions particularly elaboration tolerant, as the following results show. (Proofs of all results at www.ucl.ac.uk/slais/rob-miller/modular-e/lpnmr05long.pdf).

Definition 23 (Pre- and Post-observation/action Points). *Given a domain description D, a **post-observation point** of D is a time-point T_p such that, for every h-proposition of the form "ϕ* `holds-at` *T" in D, $T \preceq T_p$. A **pre-action point** (respectively **post-action point**) is a time-point T_a such that, for every o-proposition "A* `occurs-at` *T" in D, $T_a \preceq T$ (respectively $T_a \succeq T$).*

Definition 24 (Projection Domain Description). *The domain description D is a **projection domain description** if there exists a time-point which is both a post-observation point and a pre-action point of D.*

Theorem 1 (Free Will Theorem). *Let M be a model of a finite domain description D, let O be a finite set of o-propositions, and let T_n be a time-point which is both a post-observation point for D and a pre-action point for O. Then there is a model M_O of $D \cup O$ such that for any fluent F and time-point $T \preceq T_n$, $M_O(F,T) = M(F,T)$.*

Corollary 1 (Free Will Corollary). *Let D and D' be domain descriptions and let T_n be a post-observation point for both D and D'. Let D and D' differ only by o-propositions referring to time-points greater than or equal to T_n and let M be a model of D. The there is a model M' of D' such that $M(F,T) = M'(F,T)$ for all fluent constants F and all time-points T such that $T \preceq T_n$.*

Corollary 2 (Action Elaboration Tolerance Corollary). *Let D be a consistent domain description and let O be a finite set of o-propositions. If there exists a time-point T_n which is both a post-observation point for D and a pre-action point for O, then $D \cup O$ is consistent.*

Theorem 2 demonstrates the robustness and elaboration tolerance of $\mathcal{ME}$ theories by showing that their consistency is contingent only on the internal consistency of the static laws and on whether observations match with predicted effects.

Theorem 2 (Theorem of Causal Elaboration Tolerance). *Let D_a be a consistent domain description consisting only of a-propositions and let E be a finite set of o-, c- and p-propositions. Then $D_a \cup E$ is also a consistent domain description.*

Lack of space prevents us from giving further formal results here on the link between global and local qualifications as illustrated in examples 3 and 4. These results show their complex relationship and hence the difficulty to have modularity when a framework relies overly on explicit local qualification.

5 Exogenous Qualifications

$\mathcal{ME}$'s base semantics offers an elaboration tolerant solution to the endogenous qualification problem, where properties of the domain implicitly qualify the effect laws. It is, nonetheless, still possible that an effect fails to be produced as expected. Such a scenario occurs, for instance, when we elaborate Example 2 by observing the car not running at time 2. No known reason can explain this unexpected observation, so it needs to be attributed to an exogenous cause.

A way to reconcile such conflicts is to assume that every effect law of a domain description is implicitly qualified [4] by a set of extra preconditions, written *Normal_exo* that symbolizes the normal conditions under which the law operates successfully. These preconditions are outside the user's language or *exogenous* [13], and package together all the unknown conditions necessary for the effect law to successfully generate its effect. They hold true by default unless the observations in a given narrative make the domain description inconsistent.

Definition 25 (Default Domain Description). *Let D be a domain description. To obtain the* **default domain description** D_d **associated with** *D: (i) replace every c-proposition "C* `causes` *L" with "$C \cup Normal_exo(C, L)$* `causes` *L", and (ii) add the* ***n-proposition*** *"*`normally` *$norm_exo(_)$" for every fluent $norm_exo(_)$ in some set $Normal_exo(_)$.*

The exogenous fluents *norm_exo* that belong to the *Normal_exo* sets depend on assumptions on the nature of the failures of the effect law, in the particular domain of application. A meta-level **recovery policy** can be chosen a-priori appropriate for the domain at hand. Omitting the details, a recovery policy defines what other effect laws will be assumed to fail once a given effect law is observed to fail. One can define recovery policies where (i) no other effect laws are assumed to fail, (ii) all effect laws sharing the same effect L are also assumed to fail, (iii) all effect laws sharing the same event set C are also assumed to fail, etc. Irrespective of the recovery policy, the default models of domain D are given via the same simple minimization of the exogenous fluents over the (strict) models of the associated default domain D_d.

Definition 26 (Default Model). *Let D be a domain description, T_a a pre-action point of D, and D'_d the default domain description associated with D but without its n-propositions. Then, the restriction of M to fluents other than the $norm_exo(_)$ fluents is a* **default model** *of D iff (1))M is a model of D'_d, and (2) There is no model M' of D'_d such that $N \subset N'$, where*

$$N = \{norm_exo(_) \mid M(norm_exo(_), T_a) = true\},$$
$$N' = \{norm_exo(_) \mid M'(norm_exo(_), T_a) = true\}.$$

So far we have assumed that once an effect law is observed to fail, all subsequent instances will also fail by virtue of the persistence of *norm_exo* fluents. Various alternatives are also possible and the semantics can easily be adapted to support them. An observed failed effect law might, for example, cause its subsequent instances to fail nondeterministically, or not fail at all. Hence, in addition to failures, we can also have uncertain failures, or "accidents" (see [13]).

The existence of default models can be guaranteed as long as domains are a-consistent, point-wise consistent w.r.t. h-propositions, and do not violate fluent persistence. This requirement is captured by the notion of a *frame model*, which (assuming a "coupled accidents" recovery policy, where the exogenous qualification of a causal law exactly implies the exogenous qualification of all other causal laws applied to the same time-point) can be defined similarly to a model with the exception that the change set mapping $c(\cdot)$ can map arbitrary time-points to the empty set. Intuitively, this frame model definition allows all causal laws at some time-point to simply fail to produce their effects, as long as the successful production of their effects is not required to explain the change in the truth-value of some fluents.

Theorem 3 (Default Model Existence). *A domain description D has a default model iff it has a frame model.*

6 Summary and Related and Future Work

We have shown how $\mathcal{ME}$ can represent non-deterministic narrative domains involving concurrency, static laws and indirect effects. We have formally characterized $\mathcal{ME}$'s high degree of modularity and elaboration tolerance, enabled by an exceptionally full solution to the ramification problem able to deal with looping systems of indirect effects, and race conditions between competing causal laws. These properties help separate out, and provide a principled solutions to, the endogenous and exogenous qualification problems. Endogenous qualifications may be either locally specified or globally derived within the base semantics, whereas exogenous qualifications are provided by the use of default minimization.

Our approach to the qualification problem and its links to ramifications partly follows that in [13]. But $\mathcal{ME}$'s fuller solution to the frame problem, which covers both successful and failed action attempts, enables it to use the same default reasoning mechanism to deal with not just the “weak” but also the “strong” qualification problem as described in [13]. Two other important aspects in which $\mathcal{ME}$ differs from [13] are (a) the more complete treatment of ramifications, e.g. for concurrent effects and (b) the notion of global qualification which gives $\mathcal{ME}$ a higher degree of modularity. Our results are in line with the recent study of modularity in [6] which again highlights the link between modularity and free-will properties.

Our solution to the ramification problem is related to that in [1] in that the indirect effects of actions are defined constructively through causal laws. But $\mathcal{ME}$'s processed-based semantics differs in that it (a) embraces nondeterminism resulting from the possible orderings by which effects are realized, and (b) attributes meaning to domains (e.g, Example 9) that are deemed ill-formed in [1].

Irrespective of the qualification problem, the “free will” property of Theorem 1 is important to avoid *anomalous planning*, whereby unintended “plans” can be abduced or deduced for the converse of a precondition of an effect law by virtue of a lack of model for the successful application of that law. (See [11] for an example.) Although lack of space prevents an illustration here, anomalous plans are easy to construct in formalisms such as the Language C [5] which express action non-executability in terms of inconsistency. But they also arise in any framework (such as [7]) unable to provide models for all combinations of causal laws.

We currently have a prototype implementation of $\mathcal{ME}$'s base semantics in Prolog. The declarative programming style should facilitate an easy proof of the soundness and completeness of the implementation w.r.t. to $\mathcal{ME}$'s semantics. On the other hand, different techniques might be needed to address the *computational qualification problem* [3] of avoiding considering the majority of qualifications during the computation. Similar techniques might also prove useful when computing default models, where one does not want to consider all possible ways causal laws might fail, but rather deduce which ones should fail. To this end we are currently considering the use of satisfiability methods or Answer Set Programming (along the lines of [8,12,2]), as well as argumentation (or

abduction) based computational methods. We also aim to study subclasses (as in [2]) of $\mathcal{ME}$, where the computational complexity of reasoning decreases.

There are several aspects of $\mathcal{ME}$ that deserve further study. One is the extent to which static laws should be regarded as specific to the temporal granularity of the representation (how would we refine the role that a-propositions play in computing indirect effects?). A detailed comparison would also be useful on different recovery policies used in $\mathcal{ME}$'s approach to the exogenous qualification problem. We would also like to investigate the use of priority policies on different $\mathcal{ME}$ models, e.g. to prefer non-change in nondeterministic situations.

Acknowledgement. This work was partially supported by the IST programme of the EC, FET under the IST- 2001-32530 SOCS project, within the Global Computing proactive initiative.

References

1. M. Denecker, D. Theseider-Dupré, and K. Van Belleghem. An inductive definition approach to ramifications. *Linkoping Electronic Articles in Computer and Information Science*, 3(7):1–43, 1998.
2. Y. Dimopoulos, A. Kakas, and L. Michael. Reasoning about actions and change in answer set programming. In *Proceedings of LPNMR'03*, 2004.
3. C. Elkan. On solving the qualification problem. In *Extending Theories of Actions: Formal theory and practical applications, AAAI Spring Symposium*, 1995.
4. M. Ginsberg and D. Smith. Reasoning about Action II: The Qualification Problem. *AIJ*, 35, 1988.
5. E. Giunchiglia, J. Lee, V. Lifschitz, N. McCain, and H. Turner. Nonmonotonic Causal Theories. *AIJ*, 153(1–2):49–104, 2004.
6. A. Herzig and I. Varzinczak. Domain Descriptions Should be Modular. In *Proceedings of ECAI 2004*. IOS Press, 2004.
7. A. Kakas and R. Miller. Reasoning about Actions, Narratives and Ramifications. *Journal of Electronic Transactions on AI*, 1(4), 1998.
8. H. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proceedings of NCAI'96*, 1996.
9. V. Lifschitz. Missionaries and cannibals in the causal calculator. In *Proceedings of KR-2000*, 2000.
10. J. McCarthy. Elaboration tolerance. `http://www-formal.stanford.edy/jmc/elaboration` , 1999.
11. R. Miller and M. Shanahan. Some Alternative Formulations of the Event Calculus. *Lecture Notes in Artificial Intelligence*, 2408:452–490, 2002.
12. M. Shanahan and M. Witkowski. Event Calculus Planning Through Satisfiability. *J. of Logic Comp.*, 14(5):731–745, 2004.
13. M. Thielscher. The Qualification Problem: A solution to the problem of anomalous models. *AIJ*, 131(1–2):1–37, 2001.

PLATYPUS: A Platform for Distributed Answer Set Solving

Jean Gressmann[1], Tomi Janhunen[2], Robert E. Mercer[3], Torsten Schaub[1,⋆], Sven Thiele[1], and Richard Tichy[1,3]

[1] Institut für Informatik, Universität Potsdam, Postfach 900327, D-14439 Potsdam, Germany
[2] Helsinki University of Technology, Department of Computer Science and Engineering, Laboratory for Theoretical Computer Science, P.O. Box 5400, FI-02015 TKK, Finland
[3] Computer Science Department, Middlesex College, The University of Western Ontario, London, Ontario, Canada N6A 5B7

Abstract. We propose a model to manage the distributed computation of answer sets within a general framework. This design incorporates a variety of software and hardware architectures and allows its easy use with a diverse cadre of computational elements. Starting from a generic algorithmic scheme, we develop a platform for distributed answer set computation, describe its current state of implementation, and give some experimental results.

1 Introduction

The success of Answer Set Programming (ASP) has been greatly boosted by the availability of highly efficient ASP solvers [1,2]. However, its expanding range of application creates an increasing demand for more powerful computational devices. We address this by proposing a generic approach to distributed answer set solving that permits exploitation of the increasing availability of clustered and/or multi-processor machines.

We observe that the search strategies of most current answer set solvers naturally decompose into a deterministic and a non-deterministic part, borrowing from the well-known DPLL satisfiability checking algorithm [3]. While the non-deterministic part is usually realized through heuristically driven *choice* operations, the deterministic one is normally based on advanced *propagation* operations, often amounting to the computation of Fitting's [4] or well-founded semantics [5]. Roughly, the idea is: starting with an empty (partial) assignment of truth values to atoms, successively apply propagation and choice operations, gradually extending a partial assignment, until finally a total assignment, expressing an answer set, is obtained. The overall approach is made precise in Algorithm 1, which closely follows `smodels` [1].[1,2] When called with SMODELS$((\emptyset,\emptyset))$, it computes all answer sets of a logic program via backtracking. A partial assignment is represented as a pair (X,Y) of sets of atoms, in which X and Y contain those atoms assigned true and false, respectively. Informally, propagation is done with the EXPAND function (in Line 1); choices are done with CHOOSE (in Line 4).

⋆ Affiliated with the School of Computing Science at Simon Fraser University, Burnaby, Canada.
[1] `dlv` works analogously yet tuned to disjunctive programs.
[2] We use `typewriter` font when referring to actual systems.

C. Baral et al. (Eds.): LPNMR 2005, LNAI 3662, pp. 227–239, 2005.
© Springer-Verlag Berlin Heidelberg 2005

The first **if**-statement accounts for invalid assignments indicating an inconsistency (in Line 2), and the second, for the case of a total assignment representing an answer set (in Line 3). Otherwise, a case-analysis is performed on the chosen atom,[3] assuming it to be true in Line 5 and false in Line 6, respectively.

Algorithm 1: SMODELS

Global : A logic program Π over alphabet $\mathcal{A}$.
Input : A partial assignment (X, Y).
Output : Print all answer sets of $\Pi \cup \{\leftarrow \textit{not}\ A \mid A \in X\} \cup \{\leftarrow A \mid A \in Y\}$.

```
  begin
1   (X', Y') ← EXPAND((X, Y))
2   if X' ∩ Y' ≠ ∅ then return
3   if X' ∪ Y' = A then
        print X'
        return
4   A ← CHOOSE(A \ (X' ∪ Y'))
5   SMODELS((X' ∪ {A}, Y'))
6   SMODELS((X', Y' ∪ {A}))
  end
```

Our approach takes advantage of this idea by relying on an encapsulated module for propagation. For sake of comparability, this is currently embodied by `smodels`' expansion procedure. Unlike `smodels`, however, we are interested in distributing parts of the search space, as invoked by the two recursive calls in Algorithm 1. To this end, we propose a general approach, based on pioneering work in distributed tree search, that accommodates a variety of different architectures for distributing the search for answer sets over different processes and processors, respectively. Distributed tree search in ASP solvers [6,7,8] has been significantly influenced by the general-purpose backtracking package, DIB [9], the culmination of a decade of research in distributed tree search. Also, much work has been carried out in the area of parallel logic programming, among which our work is particularly analogous to or-parallelism; see [10,11] for surveys of this field. However, an important difference is that concurrent prolog implementations seek to parallelize query evaluation, whereas our goal is to distribute the search for answer sets. The latter is more closely related to distributed satisfiability checking (see e.g. [12,13]), although differing in the sense that it typically suffices for satisfiability checking to find only one satisfying assignment. An early attempt to compute answer sets in parallel was made in [14] by using the model generation theorem prover MGTP.

We start by developing an iterative enhancement of Algorithm 1 that is based on an explicit representation of the search space. Whenever the system environment allows us to delegate a part of this search space, it may be transferred to another computational device. Although our early efforts have focused on `smodels` as the computing engine, we differ from [7] and [8] in that their design philosophy is to build distributed versions of `smodels`, whereas our approach (1) modularizes (and is thus independent of) the propagation engine, and (2) incorporates a flexible distribution scheme, accommodating different distribution policies and distribution architectures, for instance. Regarding

[3] In fact, `smodels` chooses a literal, thereby dynamically deciding the order between the two calls in Line 5 and Line 6.

the latter, the current system supports a multiple process (by forking) and a multiple processor (by MPI [15]) architecture. A multi-threaded variant is currently under development. The multi-process and multi-threaded architectures can also be run on a multi-processor environment to achieve real speed-ups (compared to a one-processor environment).

2 Definitions and Notation

A *logic program* is a finite set of rules of the form

$$p_0 \leftarrow p_1, \ldots, p_m, not\, p_{m+1}, \ldots, not\, p_n \ , \tag{1}$$

where $n \geq m \geq 0$, and each p_i $(0 \leq i \leq n)$ is an *atom* in some alphabet $\mathcal{A}$. Given a rule r as in (1), we let $head(r)$ denote the *head* (set), $\{p_0\}$, of r and $body^+(r) = \{p_1, \ldots, p_m\}$ and $body^-(r) = \{p_{m+1}, \ldots, p_n\}$, the sets of positive and negative *body* literals, respectively. Also, we allow for *integrity constraints*, where $head(r) = \emptyset$. The *reduct*, Π^X, of a program Π *relative to* a set X of atoms is defined as

$$\Pi^X = \{head(r) \leftarrow body^+(r) \mid r \in \Pi, body^-(r) \cap X = \emptyset\} \ .$$

Then,[4] a set X of atoms is an *answer set* of a program Π if X is a $\subseteq$–minimal model of Π^X. We use $AS(\Pi)$ for denoting the set of all answer sets of a program Π.

As an example, consider program Π, consisting of rules

$$\begin{array}{lll} p \leftarrow not\, q, & r \leftarrow p, & s \leftarrow r, not\, t, \\ q \leftarrow not\, p, & r \leftarrow q, & t \leftarrow r, not\, s. \end{array} \tag{2}$$

Program Π has 4 answer sets, $\{p, r, s\}$, $\{p, r, t\}$, $\{q, r, s\}$, and $\{q, r, t\}$. Adding integrity constraint $\leftarrow q, r$ eliminates the two last sets.

For computing answer sets, we rely on *partial assignments*, mapping atoms in $\mathcal{A}$ onto true, false, or undefined. We represent such assignments as pairs (X, Y) of sets of atoms, in which X contains all true atoms and Y all false ones. An answer set X is then represented by the total assignment $(X, \mathcal{A} \setminus X)$. In general, a partial assignment (X, Y) aims at capturing a subset of the answer sets of a program Π, viz.

$$AS_{(X,Y)}(\Pi) = \{Z \in AS(\Pi) \mid X \subseteq Z, Z \cap Y \neq \emptyset\} \ .$$

3 The PLATYPUS Approach

A key observation leading to our approach is that once the program along with its alphabet is fixed, the outcome of Algorithm 1 depends only on the partial assignment given as **Input**. As made precise in the **Output** field, the resulting answer sets are then uniquely determined. Moreover, partial assignments provide a straightforward way for

[4] We use this definition since it easily includes integrity constraints. Note that any integrity constraint $\leftarrow body(r)$ can be expressed as $x \leftarrow body(r), not\, x$ by using a new atom x.

partitioning the search space, as witnessed by Lines 5 and 6 in Algorithm 1. In fact, incompatible partial assignments represent different parts of the search space.

For illustration, let us compute the answer sets of Program Π, given in (2). Starting with SMODELS$((\emptyset,\emptyset))$ forces us to choose immediately among the undefined atoms in $\{p,q,r,s,t\}$ because $(\emptyset,\emptyset)$ cannot be extended by EXPAND. Choosing p makes us call SMODELS first with $(\{p\},\emptyset)$ and then with $(\emptyset,\{p\})$. This case-analysis partitions the search space into two subspaces, the one containing all assignments making p true and the other with all assignments making p false. Following up the first call with $(\{p\},\emptyset)$, the latter gets expanded to $(\{p,r\},\{q\})$ before a choice must be made among $\{s,t\}$. Again, the search space becomes partitioned, and we obtain two total assignments, $(\{p,r,s\},\{q,t\})$ and $(\{p,r,t\},\{q,s\})$, whose first components are printed as answer sets. The two remaining answer sets are obtained analogously, but with the roles of p and q interchanged.

For distributing the computation of answer sets, the idea is to decompose the search space by means of partial assignments. For instance, instead of invoking a single process via SMODELS$((\emptyset,\emptyset))$, we may initiate two independent ones by calling SMODELS on $(\{p\},\emptyset)$ and $(\emptyset,\{p\})$, possibly even on different machines. Although this static distribution of the search space results in a fair division of labor, such a balance is hardly achievable in general. To see this, consider the choice of r instead of p, resulting in calling SMODELS with $(\{r\},\emptyset)$ and $(\emptyset,\{r\})$. While the former process gets to compute all 4 answer sets, the latter terminates almost immediately, since the EXPAND function yields an invalid assignment.[5] In such a case a dynamic redistribution of the search space is clearly advantageous. That is, once the second process is terminated, the first one may delegate some of its remaining tasks to the second one.

To this end, we propose a general approach that accommodates a variety of different modes for distributing the search for answer sets over different processes and/or processors. We start by developing an iterative enhancement of Algorithm 1 that is based on an explicit representation of the search space in terms of partial assignments. Algorithm 2 gives our generic PLATYPUS[6] algorithm for distributed answer set solving. A principal goal in its design is to allow for as much *generality* as possible. Specific instances contain trade-offs, for example, arbitrary access to the search space versus compact spatial representations of it. Another major design goal is *minimal communication* in terms of message size. To this end, PLATYPUS relies on the omnipresence of the given logic program Π along with its alphabet $\mathcal{A}$ as global parameters. Communication is limited to passing partial assignments as representatives of parts of the search space.

The only input variable S delineates the initial search space given to a specific instance of PLATYPUS. S is thus a set of partial assignments over alphabet $\mathcal{A}$. Although this explicit representation offers an extremely flexible access to the search space, it must be handled with care since it grows exponentially in the worst case. Without Line 9, Algorithm 2 computes all answer sets in $\bigcup_{(X,Y)\in S} AS_{(X,Y)}(\Pi)$, or equivalently,

[5] This nicely illustrates that the choice of the branching atom is more crucial in a distributed setting.

[6] *platypus*, small densely furred aquatic monotreme of Australia and Tasmania having a broad bill and tail and webbed feet.

Algorithm 2: PLATYPUS

```
Global  : A logic program Π over alphabet 𝒜.
Input   : A nonempty set S of partial assignments.
Output  : Print a subset of the answer sets of Π (cf. (3)).

  repeat
1   (X, Y) ← CHOOSE(S)
2   S ← S \ {(X, Y)}
3   (X', Y') ← EXPAND((X, Y))
4   if X' ∩ Y' = ∅ then
5       if X' ∪ Y' = 𝒜 then
6           print X'
        else
7           A ← CHOOSE(𝒜 \ (X' ∪ Y'))
8           S ← S ∪ { (X' ∪ {A}, Y'), (X', Y' ∪ {A}) }
9       S ← DELEGATE(S)
  until S = ∅
```

$$\bigcup_{(X,Y)\in S} AS(\Pi \cup \{\leftarrow \mathit{not}\ A \mid A \in X\} \cup \{\leftarrow A \mid A \in Y\})\ . \tag{3}$$

With Line 9, a subset of this set of answer sets is finally obtained (from a specific PLATYPUS instance). Clearly, depending on which parts of the search space are removed by delegation (see below), this algorithm is subject to incomplete and redundant search behaviour, unless an appropriate delegation strategy is used.

A PLATYPUS instance iterates until its local search space has been processed. Before detailing the loop's body, let us fix the formal behavior of the functions and procedures used by PLATYPUS (in order of appearance).

CHOOSE: Given a set[7] X, CHOOSE(X) gives some $x \in X$.

EXPAND: Given a partial assignment (X, Y), EXPAND$((X, Y))$ computes a partial assignment (X', Y') such that

1. $X \subseteq X'$ and $Y \subseteq Y'$,
2. $AS_{(X',Y')}(\Pi) = AS_{(X,Y)}(\Pi)$,
3. if $X' \cap Y' = \emptyset$ and $X' \cup Y' = \mathcal{A}$, then $AS_{(X',Y')}(\Pi) = \{X'\}$,

and furthermore (X', Y') can be closed under propagation principles such as those based on well-founded semantics and contraposition [1].[8]

DELEGATE: Given a set X, DELEGATE(X) returns a subset $X' \subseteq X$.

Both functions CHOOSE and DELEGATE are in principle non-deterministic selection functions. As usual, CHOOSE is confined to a single element, whereas DELEGATE selects an entire subset. In sum, PLATYPUS adds two additional sources of non-determinism. While the one in Line 1 is basically a "*don't care*" choice, the one in Line 9 must be handled with care since it removes assignments from the local search space.

[7] The elements of X are arbitrary in view of Lines 1 and 7.

[8] In practice, propagation may even go beyond well-founded semantics, as for instance with `smodels`'s lookahead.

The EXPAND function hosts the deterministic part of Algorithm 2; it is meant to be accomplished by an off-the-shelf system that is used as a black-box providing both sufficiently firm as well as efficient propagation operations which aim to reduce the remaining local search space resulting from choice operations.

DELEGATE permits some answer set computation tasks embodied in S to be assigned to other processes and/or processors. The assignments returned in Line 9 have not been delegated and thus remain in S. The removed assignments are either dealt with by other PLATYPUS instances or even algorithms other than PLATYPUS. The elimination of search space constituents is a delicate operation insofar as we may lose completeness or termination. For example, an implementation of DELEGATE that does not reassign all removed constituents is incomplete. Accordingly, passing certain constituents around forever would lead to non-termination.[9]

For illustration, we present some concrete specifications of DELEGATE, given in Algorithms 3 and 4. The common idea is that the system wide number of PLATYPUS

Algorithm 3: DELEGATE$_1$

```
Global  : Two integers k, n indicating the current and maximum number of PLATYPUS
          instances.
Input   : A set S of partial assignments.
Output  : A subset of S.
begin
    while (k < n) ∧ (S ≠ ∅) do
        (X, Y) ← CHOOSE(S)
        S ← S \ {(X, Y)}
        k ← k + 1
        distribute PLATYPUS({(X, Y)})
    return S
end
```

Algorithm 4: DELEGATE$_2$

```
Global  : Two integers k, n indicating the current and maximum number of PLATYPUS
          instances.
Input   : A set S of partial assignments.
Output  : A subset of S.
begin
    if k < n then
        (S, D) ← SPLIT(S)
        k ← k + 1
        distribute PLATYPUS(D)
    return S
end
```

instances is limited (by n). Variable k holds the current number of PLATYPUS instances. Accordingly, in this specific setting, k must be declared in Algorithm 2 as a global variable and decremented after each execution of the **repeat** loop. In Algorithm 3, the delegation procedure tries to maximize the global number of PLATYPUS instances. Without

[9] In fact, this cannot happen in a pure PLATYPUS setting, since at least one element is removed by each PLATYPUS instance in Line 2.

external interference (a changing k), DELEGATE$_1$ tries to produce $(n-k)$ new PLATYPUS instances. Each instance is created[10] following one of a variety of strategies, for instance, taking into account temporal or structural criteria on the partial assignments in S as well as system-specific balance criteria. Algorithm 4 is less greedy insofar as it "removes"[11] a subset from S and creates only a single new PLATYPUS instance. As with CHOOSE previously, SPLIT can be guided by various strategies, e.g. trying to share the remaining search space equally among processes/processors in order to minimize communication costs that result from delegation operations. Also, numerous mixtures of both strategies can be envisaged. Both delegation procedures guarantee completeness. To see this, it is enough to observe that every assignment and thus every part of the search space is investigated by one PLATYPUS instance in one way or another. Also, duplicate solutions are avoided by having exactly one solver investigate each part of the search space.

Table 1. Three PLATYPUS instances computing the answer sets of Π_2

k/n	PLATYPUS I	PLATYPUS II	PLATYPUS III
$1/3$	$\{\underline{(\emptyset,\emptyset)}\}$ $(\emptyset,\emptyset)$ $\{(\{\underline{s}\},\emptyset),\overline{(\emptyset,\{\underline{s}\})}\}$		
$2/3$	$\{\underline{(\{s\},\emptyset)}\}$ $(\{s,r\},\{t\})$ $\{(\{s,r,\underline{q}\},\{t\}),\overline{(\{s,r\},\{t,\underline{q}\})}\}$	$\{\underline{(\emptyset,\{s\})}\}$ $(\emptyset,\{s\})$ $\{(\{\underline{p}\},\{s\}),(\emptyset,\{s,\underline{p}\})\}$	
$3/3$	$\{\underline{(\{s,r,q\},\{t\})}\}$ $(\boldsymbol{\{s,r,q\}},\{t,p\})$ $\emptyset$	$\{\underline{(\{p\},\{s\})},(\emptyset,\{s,p\})\}$ $(\boldsymbol{\{p,r,t\}},\{s,q\})$ $\{(\emptyset,\{s,p\})\}$	$\{\underline{(\{s,r\},\{t,q\})}\}$ $(\boldsymbol{\{s,r,p\}},\{t,q\})$ $\emptyset$
$1/3$		$\{\underline{(\emptyset,\{s,p\})}\}$ $(\boldsymbol{\{r,t,q\}},\{s,p\})$ $\emptyset$	
$0/3$			

To illustrate, let us compute the answer sets of the program given in (2) in an environment with at most 3 PLATYPUS instances, using the delegation procedure in Algorithm 4. The **distribute** procedure is used for creating new processes. Our instance of SPLIT transfers $\lfloor |S|/2 \rfloor$ assignments to another PLATYPUS instance. These assignments are overlined in Table 1, while the ones chosen in Lines 1 and 7 in Algorithm 2 are underlined. The **print** of an answer set (in Line 5) is indicated by boldface letters. A cell in Table 1 represents an iteration in Algorithm 2. In each cell the first entry is S after Line 1, the second presents the result of the EXPAND function, and the last one is S before Line 9. Once the environment has been initialized, setting $n=3$ and $k=1$ among other things, the first PLATYPUS instance is invoked with $\{(\emptyset,\emptyset)\}$. After the first iteration, its search space contains $(\{s\},\emptyset)$, while $(\emptyset,\{s\})$ is used to create a

[10] To be precise, MPI instances are merely reinitialized; they live throughout the computation.

[11] Given a set X, SPLIT(X) returns a partition (X_1, X_2) of X.

second instance of PLATYPUS. Also, variable k is incremented by DELEGATE$_2$. After one more iteration, each PLATYPUS instance could potentially create yet another instance. Since the maximum number of processes is limited to 3, only one of them is able to create a new PLATYPUS instance. Once this is done k equals 3, which prevents DELEGATE$_2$ from creating any further processes. In our case, PLATYPUS I manages to delegate $(\{s, r\}, \{t, q\})$ while blocking any delegation by PLATYPUS II. The three processes output their answer sets. Whereas the first and third terminate, having emptied their search spaces, the second one iterates once more to compute the fourth and last answer set. (No delegation is initiated since $\lfloor |S|/2 \rfloor = \lfloor 1/2 \rfloor = 0$.)

4 The `platypus` Platform

Current technology provides a large variety of software and hardware mechanisms for distributed computing. Among them, we find single- and multi-threaded processes, multiple processes, as well as multiple processors, sometimes combined with multiple processes and threads.

The goal of the `platypus` platform is to provide an easy and flexible use of these architectures for ASP. To begin with, we have implemented a multiple process and a multiple processor variant of `platypus`. A multi-threaded variant is currently under development. To enable the generality of the approach, the `platypus` system is designed in a strictly modular way. The central module consists of a *black-box* providing the functionality of the EXPAND function. This module provides a fixed interface that permits wrapping different off-the-shelf propagation engines. A second module deals with the search space given by variable S in Algorithms 2, 3, and 4. Last but not least, distribution is handled by a dedicated control module fixing the respective implementation of the **distribute** operation. That is, this module controls forking, threading or MPI. With this module, each variant is equipped with a common set of distribution policies; currently all of them are realized through controllers being variations of DELEGATE$_2$ (cf. Algorithm 4), where SPLIT is replaced by CHOOSE. Hence, the currently implemented policies vary the strategy of the CHOOSE operation. Each policy depends upon the underlying distribution capabilities of the software and hardware architectures.

The multiple process variant is implemented with the `UNIX fork` mechanism, which creates a child process managed by the same operating system that controls the parent process. The forking policy requires a local controller in each process to perform the delegation task. Communication among the processes is accomplished via shared memory. The forking policy provides an easily manageable framework to test design alternatives and to experiment with low-level distribution policy decisions, such as resource saturation caused by having too many processes. The multiple processor variant runs on a cluster and relies on *The Message Passing Interface* (MPI [15]) library to perform the distribution to a process controlled by another operating system on a distinct processor. As with forking, we rely on a local controller in each process to control the delegation task; in addition, we use a global controller to manage the processes on the separate processors and the communication among processes.

To reduce the size of partial assignments and thus of passed messages, we follow [8] in storing only atoms whose truth values were assigned by a choice oper-

ation (cf. atom A in Lines 7 and 8 of Algorithm 2). Given an assignment (X, Y) along with the subsets $X_c \subseteq X$ and $Y_c \subseteq Y$ of atoms treated in this way, we have $(X, Y) = \text{EXPAND}((X_c, Y_c))$. Accordingly, some care must be taken when implementing the tests in Lines 4 and 5. To this end, the current design foresees two signals provided by the EXPAND module. The search space module (1) must support multiple access modes for accommodating the varying choice policies in Lines 1 and 9 (or better the subsequent delegation procedures) of Algorithm 2, and (2) must be handled with care, since it may grow exponentially without appropriate restrictions. So that, at this stage of the project, we can focus on issues arising from our distribution-oriented setting, the current implementation is based on the design decision that a non-distributing `platypus` instance must correspond to a traditional solver, as given in Algorithm 1. This has the advantage that we obtain a "bottom-line" solver instance that we can use for comparison with state-of-the-art solvers as well as all distributed `platypus` instances for measuring the respective trade-offs. To this end, we restrict the search space (in S) to a single branch of the search tree and implement the "local" choice operation in Line 1 of Algorithm 2 through a LIFO strategy. In this way, the "local" view of the search space can be realized by stack-based operations. Unlike this, the second access to the search space, described in the delegation procedures 3 and 4 is completely generic. This allows us to integrate various delegation policies (cf. Section 5). Following the above decision, our design also foresees the option of using a choice proposed by a given EXPAND module for implementing Line 7 in Algorithm 2, provided that this is supported by the underlying propagation engine.

Finally, let us detail some issues of the current implementation. `platypus` is written in `C++`. Its major EXPAND module is based on the `smodels` API. This also allows us to take advantage of `smodels`' heuristics in view of Line 7 in Algorithm 2 (see above). Accordingly, this module requires `lparse` for parsing logic programs. All experiments reported in Section 5 are conducted with this implementation of EXPAND. However, for guaranteeing modularity, we have also implemented other EXPAND modules, among them the one of the `nomore++` system [16]. While the distribution architecture of a `platypus` instance must be fixed at compile time, the respective parameters, like constant n in Algorithm 3, or delegation policies, such as the kind of the delegated choice point, are set via command line options at run-time. More details are given in the experimental section.

5 Experimental Results

To illustrate the feasibility of our approach, we present in Tables 2 and 3 a selection of experimental results obtained with the multi-process and the multi-processor versions of `platypus`. As a point of reference, we mention that on all these tests `smodels` is on average 1.62 times faster than the multi-process version of `platypus` limited to one process. Apart from `platypus`' early stage of development, a certain overhead is created by "double bookkeeping" due to the strict encapsulation of the EXPAND module. In this way, we trade-off some speed for our modular design.

The multi-process data were generated using the forking architecture limited to 1 to 4 active processes on a quad processor under `Linux`, comprised of 4 Opteron

Table 2. Data obtained for computing *all* answer sets with the multi-process version of `platypus`

`platypus`	`-p1 -ls`	`-p2 -ls`	`-p3 -ls`	`-p4 -ls`
color-5-10	6.44 (0)	3.54 (21.1)	2.55 (44.7)	2.12 (69.2)
color-5-15	349.07 (0)	178.70 (36.5)	120.24 (63.2)	91.15 (99.8)
hamcyc-8	4.35 (0)	2.61 (30.9)	1.94 (55.4)	1.66 (87.1)
hamcyc-9	105.88 (0)	54.59 (40.6)	36.73 (88.3)	28.13 (134.8)
pigeon-7-8	1.92 (0)	1.60 (38.5)	1.33 (68.5)	1.22 (92.7)
pigeon-7-9	7.44 (0)	4.49 (45.2)	3.35 (84.8)	2.83 (115.8)
pigeon-7-10	24.21 (0)	12.86 (49.0)	9.04 (99.3)	7.24 (142.6)
pigeon-7-11	71.40 (0)	34.93 (55.9)	23.73 (111.8)	18.34 (165.2)
pigeon-7-12	177.02 (0)	85.71 (60.5)	57.53 (124.2)	44.04 (193.3)
pigeon-8-9	18.83 (0)	9.99 (46.3)	7.09 (94.7)	5.73 (138.8)
pigeon-8-10	87.45 (0)	43.23 (49.5)	29.22 (112.3)	22.33 (163.4)
pigeon-9-10	227.72 (0)	107.14 (60.3)	71.14 (123.8)	53.56 (189.2)
schur-11-5	1.50 (0)	1.15 (18.1)	0.82 (25.4)	0.71 (34.9)
schur-12-5	5.26 (0)	3.09 (19.6)	2.23 (34.9)	1.80 (48.0)
schur-13-5	22.53 (0)	11.78 (20.6)	8.07 (37.9)	6.22 (56.9)
schur-14-5	74.51 (0)	37.80 (19.9)	25.60 (46.0)	19.25 (68.0)
schur-14-4	2.93 (0)	1.88 (16.4)	1.52 (41.9)	1.17 (46.0)
schur-15-4	8.02 (0)	4.54 (20.4)	3.23 (41.3)	2.55 (55.5)
schur-16-4	14.14 (0)	7.64 (24.1)	5.28 (43.9)	4.13 (62.3)
schur-17-4	32.50 (0)	16.92 (22.9)	11.50 (48.1)	8.82 (68.3)
schur-18-4	62.72 (0)	31.23 (21.8)	20.77 (52.8)	15.75 (74.7)
schur-19-4	132.30 (0)	65.99 (22.6)	44.02 (54.2)	33.24 (77.7)
schur-20-4	164.24 (0)	80.70 (26.1)	53.61 (60.5)	40.09 (75.0)

2.2GHz processors with 8 GB shared RAM. The multi-processor tests ran on a cluster of 5 Pentium III 1GHz PCs under `Linux` with 1 GB RAM each, with 1 to 4 active nodes and one extra node serving as master.[12] All of our timing results reflect the average elapsed time (in seconds) of the launching process/processor, respectively, over 20 runs, each computing *all* answer sets. Timing excludes parsing and printing time. Similarly, the number in parentheses indicates the average number of forks/messages passed, respectively.

The first column of Tables 2 and 3 lists the benchmarks, largely taken from the benchmarking site at [17]. Columns 2 to 5 give the forking architecture results, and columns 6 to 9 contain the data obtained from the cluster using MPI. The first row provides the command line options with which `platypus` was invoked. The `-p` option indicates the maximum number of processes or processors, respectively (n in Algorithms 3 and 4). And `-l` stands for the delegation policy. All listed tests are run in `s`*hallow* mode, delegating the smallest among all of the delegatable partial assignments available to the delegation procedure. The current system also supports a `d`*eep*, `m`*iddle*, and `r`*andom* mode. Delegating small partial assignments is theoretically reasonable since they represent the putatively largest parts of the search space, thus each delegated

[12] Note that the former processor type is much faster than the latter.

Table 3. Data obtained for computing *all* answer sets with the multi-processor version of `platypus`

`platypus`	`-p 1 -l s`	`-p 2 -l s`	`-p 3 -l s`	`-p 4 -l s`
color-5-10	28.18 (4)	14.57 (119.0)	9.99 (245.1)	7.74 (385.4)
color-5-15	1632.91 (4)	821.83 (120.3)	549.68 (258.4)	413.75 (616.5)
hamcyc-8	16.59 (4)	8.89 (146.8)	6.10 (312.4)	4.77 (454.3)
hamcyc-9	407.43 (4)	202.16 (219.8)	135.96 (585.8)	102.80 (993.6)
pigeon-7-8	6.37 (4)	4.01 (82.8)	3.23 (176.6)	2.87 (243.3)
pigeon-7-9	25.53 (4)	13.99 (131.8)	10.18 (251.6)	8.54 (349.1)
pigeon-7-10	84.28 (4)	42.92 (167.3)	30.08 (335.3)	24.01 (546.9)
pigeon-7-11	238.62 (4)	117.46 (198.3)	81.07 (476.1)	62.73 (750.0)
pigeon-7-12	590.83 (4)	291.03 (219.3)	197.33 (581.1)	150.43 (972.6)
pigeon-8-9	62.70 (4)	32.22 (146.3)	22.71 (281.8)	18.44 (486.0)
pigeon-8-10	282.19 (4)	139.72 (176.8)	95.51 (453.8)	73.75 (753.7)
pigeon-9-10	695.56 (4)	341.25 (214.0)	230.35 (586.1)	175.02 (1024.1)
schur-11-5	5.45 (4)	3.19 (40.5)	2.67 (83.7)	2.27 (133.6)
schur-12-5	18.85 (4)	9.88 (30.3)	7.19 (81.9)	5.95 (195.6)
schur-13-5	78.90 (4)	40.16 (48.5)	27.36 (104.5)	21.30 (247.5)
schur-14-5	254.78 (4)	129.08 (70.5)	86.70 (106.8)	66.05 (297.1)
schur-14-4	10.26 (4)	5.55 (27.5)	4.38 (112.9)	3.77 (187.2)
schur-15-4	27.83 (4)	14.56 (44.5)	10.24 (102.7)	8.52 (251.3)
schur-16-4	48.56 (4)	24.92 (56.3)	17.27 (122.8)	14.16 (291.6)
schur-17-4	113.29 (4)	57.61 (65.8)	39.47 (164.9)	30.36 (291.3)
schur-18-4	206.20 (4)	103.74 (47.5)	70.19 (172.5)	53.92 (369.6)
schur-19-4	450.43 (4)	225.75 (75.3)	151.40 (226.6)	113.60 (306.2)
schur-20-4	539.21 (4)	270.21 (70.0)	180.97 (237.8)	135.70 (335.9)

`platypus` instance will be given the largest task to perform, hence minimizing the amount of delegation. Our early experiments and similar observations reported in [8] support this view. In fact, selecting the largest assignment (via option `-l d`) results in much more forking/message passing and much poorer performance. For instance, using the forking architecture, we get on average for schur-20-4 with options `-p 4 -l d` a time of 155.43s and a count of 322 forks.

The results in Tables 2 and 3 are indicative of what is expected from distributed computation. When looking at each benchmark, the forking and MPI experiments show a qualitatively consistent 2-, 3-, and 4-times speed-up when doubling, tripling, and quadrupling the number of processors, with only minor exceptions. The more substantial the benchmark, the more clear-cut the speed-up. A more global and more quantitative sense of the speed-up is provided by the sum of times[13] for all benchmarks for each `-p` setting compared to "`-p 1`". These ratios are: 1, 1.98, 2.93, 3.85, and 1, 1.99, 2.96, 3.88 for forking and MPI, respectively. With this in mind, we observe on computationally undemanding benchmarks, like hamcyc-8, pigeon-7-8, or schur-11-5 no real gain. In fact, our overall experiments show that the less substantial the benchmark, the more

[13] Of course, these ratios are biased by the more substantial benchmarks, but these are the more reasonable indicators of the speed-up.

insignificant the speed-up as the overhead caused by distribution dominates the actual search time. Similarly, the sum of messages increases from "-p 2" to "-p 3" and to "-p 4" by 2.03, 2.95, and by 2.41, 4.16, respectively for forking and MPI. To interpret the number of messages in Tables 2 and 3, the one in the forking results reflects the number of delegations, whereas the number of messages in our MPI setting include $2(n+1)$ start-up and shut-down messages plus 5 handshaking messages for each delegation.

6 Summary

Conceptually, the PLATYPUS approach offers a general and flexible framework for managing the distributed computation of answer sets, incorporating a variety of different software and hardware architectures for distribution. The major design decisions were to minimize the number and size of the messages passed by appeal to partial assignments and to abstract from a serial ASP propagation system. The latter allows us to take advantage of efficient off-the-shelf engines, developed within the ASP community for the non-distributive case.

Meanwhile, the `platypus` system furnishes a platform for implementing various forms of distribution. All versions of `platypus` share the same code, except for the control module matching the specific distribution architecture. The current system supports a multiple process architecture, using the UNIX forking mechanism, and a multiple processor architecture, running on a cluster via MPI. A multi-threaded variant is currently under development. Moreover, `platypus` supports different distribution policies, being open to further extensions through well-defined interfaces.

Finally, the encouraging results from our experiments suggest that our generic approach to the distributed computation of answer sets offers a powerful computational enhancement to classical answer set solvers. In particular, we have seen a virtually optimal speed-up on substantial benchmarks, that is, the speed-up nearly matched the number of processes or processors, respectively.

The `platypus` platform is freely available on the web [18]. The current system provides us with the necessary infrastructure for manifold future investigations into distributed answer set solving. This concerns the whole spectrum of different instances of the procedures CHOOSE and SPLIT, on the one side, and **distribute** on the other. A systematic study of different options in view of dynamic load balancing will be a major issue of future experimental research. In fact, the given set of benchmarks was chosen as a representative selection demonstrating the feasibility of our approach. In view of load balancing, it will be interesting to see how the type of benchmark (and thus the underlying search space) is related to specific distribution schemes. Also, we have so far concentrated on finding *all* answer sets of a given program. When extending the system for finding some answer set(s) only, the structure of the search space will become much more important.

Acknowledgments. The first, fourth, and fifth authors were supported by DFG under grant SCHA 550/6-4. All but the third and last authors were also funded by the EC through IST-2001-37004 WASP project. The third and last authors were funded by NSERC (Canada) and SHARCNET. Finally, we would like to thank Christine Bailey for her contribution to the very early development of `platypus`.

References

1. Simons, P., Niemelä, I., Soininen, T.: Extending and implementing the stable model semantics. Artificial Intelligence **138** (2002) 181–234
2. Leone, N., Faber, W., Pfeifer, G., Eiter, T., Gottlob, G., Koch, C., Mateis, C., Perri, S., Scarcello, F.: The DLV system for knowledge representation and reasoning. ACM Transactions on Computational Logic (2005) To appear.
3. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. Communications of the ACM **5** (1962) 394–397
4. Fitting, M.: Fixpoint semantics for logic programming: A survey. Theoretical Computer Science **278** (2002) 25–51
5. van Gelder, A., Ross, K., Schlipf, J.: The well-founded semantics for general logic programs. Journal of the ACM **38** (1991) 620–650
6. Finkel, R., Marek, V., Moore, N., Truszczynski, M.: Computing stable models in parallel. In Provetti, A., Son, T., eds.: Proceedings of AAAI Spring Symposium on Answer Set Programming, AAAI/MIT Press (2001) 72–75
7. Hirsimäki, T.: Distributing backtracking search trees. Technical report, Helsinki University of Technology (2001)
8. Pontelli, E., Balduccini, M., Bermudez, F.: Non-monotonic reasoning on beowulf platforms. In Dahl, V., Wadler, P., eds.: Proceedings of the Fifth International Symposium on Practical Aspects of Declarative Languages. (2003) 37–57
9. Finkel, R., Manber, U.: DIB — a distributed implementation of backtracking. ACM Transactions on Programming Languages and Systems **9** (1987) 235–256
10. Gupta, G., Pontelli, E., Ali, K., Carlsson, M., Hermenegildo, M.: Parallel execution of prolog programs: a survey. ACM Transactions on Programming Languages and Systems **23** (2001) 472–602
11. Chassin de Kergommeaux, J., Codognet, P.: Parallel logic programming systems. ACM Computing Surveys **26** (1994) 295–336
12. Zhang, H., Bonacina, M., Hsiang, J.: PSATO: a distributed propositional prover and its application to quasigroup problems. Journal of Logic and Computation **21** (1996) 543–560
13. Blochinger, W., Sinz, C., Küchlin, W.: Parallel propositional satisfiability checking with distributed dynamic learning. Parallel Computing **29** (2003) 969–994
14. Inoue, K., Koshimura, M., Hasegawa, R.: Embedding negation as failure into a model generation theorem prover. In Kapur, D., ed.: Proceedings of the Eleventh International Conference on Automated Deduction. Springer-Verlag (1992) 400–415
15. Gropp, W., Lusk, E., Thakur, R.: Using MPI-2: Advanced Features of the Message-Passing Interface. The MIT Press (1999)
16. (`http://www.cs.uni-potsdam.de/nomore`)
17. (`http://asparagus.cs.uni-potsdam.de`)
18. (`http://www.cs.uni-potsdam.de/platypus`)

Solving Hard ASP Programs Efficiently*

Wolfgang Faber** and Francesco Ricca

Department of Mathematics, University of Calabria,
I-87030 Rende (CS), Italy
{faber, ricca}@mat.unical.it

Abstract. Recent research on answer set programming (ASP) systems, has mainly focused on solving NP problems more efficiently. Yet, disjunctive logic programs allow for expressing every problem in the complexity classes Σ_2^P and Π_2^P. These classes are widely believed to be strictly larger than NP, and several important AI problems, like conformant and conditional planning, diagnosis and more are located in this class.

In this paper we focus on improving the evaluation of Σ_2^P/Π_2^P-hard ASP programs. To this end, we define a new heuristic h_{DS} and implement it in the (disjunctive) ASP system DLV. The definition of h_{DS} is geared towards the peculiarites of hard programs, while it maintains the benign behaviour of the well-assessed heuristic of DLV for NP problems.

We have conducted extensive experiments with the new heuristic. h_{DS} significantly outperforms the previous heuristic of DLV on hard 2QBF problems. We also compare the DLV system (with h_{DS}) to the QBF solvers SSolve, Quantor, Semprop, and yQuaffle, which performed best in the QBF evaluation of 2004. The results of the comparison indicate that ASP systems currently seem to be the best choice for solving Σ_2^P/Π_2^P-complete problems.

1 Introduction

Answer Set Programming (ASP) is a novel programming paradigm, which has been recently proposed in the area of nonmonotonic reasoning and logic programming. The idea of answer set programming is to represent a given computational problem by a logic program whose answer sets correspond to solutions, and then use an answer set solver to find such a solution [1]. The knowledge representation language of ASP is very expressive in a precise mathematical sense; in its general form, allowing for disjunction in rule heads and nonmonotonic negation in rule bodies, ASP can represent *every* problem in the complexity class Σ_2^P and Π_2^P (under brave and cautious reasoning, respectively) [2]. Thus, ASP is strictly more powerful than SAT-based programming, as it allows us to solve problems which cannot be translated to SAT in polynomial time. The high expressive power of ASP can be profitably exploited in AI, which often has to deal with

* This work was supported by the European Commission under projects IST-2002-33570 INFOMIX, and IST-2001-37004 WASP.

** Funded by an APART grant of the Austrian Academy of Sciences.

C. Baral et al. (Eds.): LPNMR 2005, LNAI 3662, pp. 240–252, 2005.
© Springer-Verlag Berlin Heidelberg 2005

problems of high complexity. For instance, problems in diagnosis and planning under incomplete knowledge are complete for the complexity class Σ_2^P or Π_2^P [3,4], and can be naturally encoded in ASP [5,6].

Most of the optimization work on ASP systems has focused on the efficient evaluation of non-disjunctive programs (whose power is limited to NP/co-NP), whereas the optimization of full (disjunctive) ASP programs has been treated in fewer works (e.g., in [7,8]). In particular, we are not aware of any work concerning heuristics for Σ_2^P/Π_2^P-hard ASP programs.

Since the model generators of ASP systems, like DLV [9] and Smodels [10], are similar to the Davis-Putnam procedure, employed in many SAT solvers, the heuristic (branching rule) for the selection of the branching literal (i.e., the criterion determining the literal to be assumed true at a given stage of the computation) is fundamentally important for the efficiency of an ASP system. Some other systems, like ASSAT [11] or Cmodels [12] use a SAT solver directly as a black box, and thus have limited means of tuning the heuristic. Also note that all of Smodels, ASSAT, and Cmodels are confined to NP problems. Since our focus is on harder problems, we will not consider these systems further.

In this paper, we address the following two questions:
▶ Can the heuristics of ASP systems be refined to deal more efficiently with Σ_2^P/Π_2^P-hard ASP programs?
▶ On hard Σ_2^P/Π_2^P problems, can ASP systems compete with other AI systems, like QBF solvers?

We define a new heuristic h_{DS} for the (disjunctive) ASP system DLV. The new heuristic aims at improving the evaluation of Σ_2^P/Π_2^P-hard ASP programs, but it is designed to maintain the benign behaviour of the well-assessed heuristic of DLV on NP problems like 3SAT and Blocks-World, on which it proved to be very effective [13]. We experimentally compare h_{DS} against the DLV heuristic on hard 2QBF instances, generated following recent works presented in the literature that describe transition phase results for QBFs [14,15]. h_{DS} significantly outperforms the heuristic of DLV on 2QBF.

To check the competitiveness of ASP w.r.t. QBF solvers on hard problems, we carry out an experimental comparison of the DLV system (with the new heuristic h_{DS}) with four prominent QBF solvers, which performed best at the 2004 QBF evaluation[16,17]: SSolve, Semprop, Quantor, yQuaffle. The results of the comparison, performed on instances used in the QBF competition and on a set of randomly generated instances for the *Strategic Companies* problem, indicate that ASP systems currently perform better than QBF systems on Σ_2^P/Π_2^P-hard problems.

2 Answer Set Programming Language

2.1 ASP Programs

A *(disjunctive) rule* r is a formula

$$a_1 \vee \cdots \vee a_n \text{ :- } b_1, \cdots, b_k, \texttt{ not } b_{k+1}, \cdots, \texttt{ not } b_m.$$

where $a_1, \cdots, a_n, b_1, \cdots, b_m$ are atoms and $n \geq 0$, $m \geq k \geq 0$. The disjunction $a_1 \vee \cdots \vee a_n$ is the *head* of r, while the conjunction $b_1, \cdots, b_k, \texttt{not}\ b_{k+1}, \cdots, \texttt{not}\ b_m$ is the *body*, $b_1, \cdots, b_k$ the *positive body*, and $\texttt{not}\ b_{k+1}, \cdots, \texttt{not}\ b_m$ the *negative body* of r.

An *(ASP) program* $\mathcal{P}$ is a finite set of rules. An object (atom, rule, etc.) is called *ground* or *propositional*, if it contains no variables.

2.2 Answer Sets

Given a program $\mathcal{P}$, let the *Herbrand Universe* $U_{\mathcal{P}}$ be the set of all constants appearing in $\mathcal{P}$ and the *Herbrand Base* $B_{\mathcal{P}}$ be the set of all possible ground atoms which can be constructed from the predicate symbols appearing in $\mathcal{P}$ with the constants of $U_{\mathcal{P}}$.

Given a rule r, $Ground(r)$ denotes the set of rules obtained by applying all possible substitutions σ from the variables in r to elements of $U_{\mathcal{P}}$. Similarly, given a program $\mathcal{P}$, the *ground instantiation* $Ground(\mathcal{P})$ of $\mathcal{P}$ is the set $\bigcup_{r \in \mathcal{P}} Ground(r)$.

For every program $\mathcal{P}$, we define its answer sets using its ground instantiation $Ground(\mathcal{P})$ in two steps: First we define the answer sets of positive programs, then we give a reduction of general programs to positive ones and use this reduction to define answer sets of general programs.

A set L of ground literals is said to be *consistent* if, for every atom $\ell \in L$, its complementary literal $\texttt{not}\ \ell$ is not contained in L. An interpretation I for $\mathcal{P}$ is a consistent set of ground literals over atoms in $B_{\mathcal{P}}$.[1] A ground literal ℓ is *true* w.r.t. I if $\ell \in I$; ℓ is *false* w.r.t. I if its complementary literal is in I; ℓ is *undefined* w.r.t. I if it is neither true nor false w.r.t. I. Interpretation I is *total* if, for each atom A in $B_{\mathcal{P}}$, either A or $\texttt{not}\ A$ is in I (i.e., no atom in $B_{\mathcal{P}}$ is undefined w.r.t. I). A total interpretation M is a *model* for $\mathcal{P}$ if, for every $r \in Ground(\mathcal{P})$, at least one literal in the head is true w.r.t. M whenever all literals in the body are true w.r.t. M. X is an *answer set* for a positive program $\mathcal{P}$ if it is minimal w.r.t. set inclusion among the models of $\mathcal{P}$.

Example 1. For the positive program $\mathcal{P}_1 = \{a \vee b \vee c.,\ :-a.\}$, $\{b, \texttt{not}\ a, \texttt{not}\ c\}$ *and* $\{c, \texttt{not}\ a, \texttt{not}\ b\}$ *are the answer sets. For the positive program* $\mathcal{P}_2 = \{a \vee b \vee c.,\ :-a.,\ b{:-}c.,\ c{:-}b.\}$, $\{b, c, \texttt{not}\ a\}$ *is the only answer set.* ∎

The *reduct* or *Gelfond-Lifschitz transform* of a general ground program $\mathcal{P}$ w.r.t. an interpretation X is the positive ground program $\mathcal{P}^X$, obtained from $\mathcal{P}$ by (i) deleting all rules $r \in \mathcal{P}$ whose negative body is false w.r.t. X and (ii) deleting the negative body from the remaining rules.

An answer set of a general program $\mathcal{P}$ is a model X of $\mathcal{P}$ such that X is an answer set of $Ground(\mathcal{P})^X$.

Example 2. Given the (general) program $\mathcal{P}_3 = \{a \vee b{:-}c.,\ b{:-}\texttt{not}\ a, \texttt{not}\ c.,\ a \vee c{:-}\texttt{not}\ b.\}$ *and* $I = \{b, \texttt{not}\ a, \texttt{not}\ c\}$, *the reduct* $\mathcal{P}_3^I$ *is* $\{a \vee b{:-}c., b.\}$. *I is an answer set of* $\mathcal{P}_3^I$, *and for this reason it is also an answer set of* $\mathcal{P}_3$. ∎

[1] We represent interpretations as set of literals, since we have to deal with partial interpretations in the next sections.

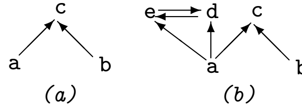


Fig. 1. *Graphs (a)* $DG_{\mathcal{P}_4}$*, and (b)* $DG_{\mathcal{P}_5}$

2.3 Some ASP Properties

Given an interpretation I for a ground program $\mathcal{P}$,we say that a ground atom A is *supported* in I if there exists a *supporting* rule $r \in ground(\mathcal{P})$ such that the body of r is true w.r.t. I and A is the only true atom in the head of r.

Proposition 1. *[18,19,20]* If M is an answer set of a program $\mathcal{P}$, then all atoms in M are supported.

Another relevant property of ASP programs is head-cycle freeness (HCF). With every ground program $\mathcal{P}$, we associate a directed graph $DG_{\mathcal{P}} = (N, E)$, called the *dependency graph* of $\mathcal{P}$, in which (i) each atom of $\mathcal{P}$ is a node in N and (ii) there is an arc in E directed from a node a to a node b iff there is a rule r in $\mathcal{P}$ such that b and a appear in the head and body of r, respectively.

The graph $DG_{\mathcal{P}}$ singles out the dependencies of the head atoms of a rule r from the positive atoms in its body.[2]

Example 3. Consider the program $\mathcal{P}_4 = \{a \vee b.,\ c{:}{-}a.,\ c{:}{-}b.\}$*, and the program* $\mathcal{P}_5 = \mathcal{P}_4 \cup \{d \vee e{:}{-}a.,\ d{:}{-}e.,\ e{:}{-}d, \texttt{not}\ b.\}$*. The dependency graph* $DG_{\mathcal{P}_4}$ *of* $\mathcal{P}_4$ *is depicted in Figure 1 (a), while the dependency graph* $DG_{\mathcal{P}_5}$ *of* $\mathcal{P}_5$ *is depicted in Figure 1 (b).* ∎

The dependency graphs allow us to single out HCF programs [21]. A program $\mathcal{P}$ is *HCF* iff there is no rule r in $\mathcal{P}$ such that two atoms occurring in the head of r are in the same cycle of $DG_{\mathcal{P}}$.

Example 4. The dependency graphs given in Figure 1 reveal that program $\mathcal{P}_4$ *of Example 3 is HCF and that program* $\mathcal{P}_5$ *is not HCF, as rule* $d \vee e \leftarrow a$ *contains in its head two atoms belonging to the same cycle of* $DG_{\mathcal{P}_5}$*.* ∎

HCF programs are computationally easier than general (non-HCF) programs.

Proposition 2. *[21,2]* 1. Deciding whether an atom belongs to some answer set of a ground HCF program $\mathcal{P}$ is NP-complete. 2. Deciding whether an atom belongs to some answer set of a ground (non-HCF) program $\mathcal{P}$ is Σ_2^P-complete.

3 Answer Set Computation

In this section, we describe the main steps of the computational process performed by ASP systems. We will refer particularly to the computational engine

[2] Note that negative literals cause no arc in $DG_{\mathcal{P}}$.

of the DLV system, which will be used for the experiments, but also other ASP systems, like Smodels, employ a very similar procedure.

An answer set program $\mathcal{P}$ in general contains variables. The first step of a computation of an ASP system eliminates these variables, generating a ground instantiation $ground(\mathcal{P})$ of $\mathcal{P}$.[3] The hard part of the computation is then performed on this ground ASP program generated by the instantiator.

Function ModelGenerator(I: Interpretation): Boolean;
begin
  I := DetCons(I);
  if I = $\mathcal{L}$ **then return** False; (* inconsistency *)
  if no atom is undefined in I **then return** IsAnswerSet(I);
  Select an undefined ground atom A according to a heuristic;
  if ModelGenerator($I \cup \{A\}$) **then return** True;
  else return ModelGenerator($I \cup \{\texttt{not}\ A\}$);
end;

Fig. 2. Computation of Answer Sets

The heart of the computation is performed by the Model Generator, which is sketched in Figure 2. Roughly, the Model Generator produces some "candidate" answer sets. The stability of each of them is subsequently verified by the function *IsAnswerSet(I)*, which verifies whether the given "candidate" I is a minimal model of the program $Ground(\mathcal{P})^I$ obtained by applying the GL-transformation w.r.t. I and outputs the model, if so. *IsAnswerSet(I)* returns True if the computation should be stopped and False otherwise.

The ModelGenerator function is first called with parameter I set to the empty interpretation.[4] If the program $\mathcal{P}$ has an answer set, then the function returns True setting I to the computed answer set; otherwise it returns False. The Model Generator is similar to the Davis-Putnam procedure employed by SAT solvers. It first calls a function DetCons(), which returns the extension of I with the literals that can be deterministically inferred (or the set of all literals $\mathcal{L}$ upon inconsistency). This function is similar to a unit propagation procedure employed by SAT solvers, but exploits the peculiarities of ASP for making further inferences (e.g., it exploits the knowledge that every answer set is a minimal model). If DetCons does not detect any inconsistency, an atom A is selected according to a heuristic criterion and ModelGenerator is called on $I \cup \{A\}$ and on $I \cup \{\texttt{not}\ A\}$. The atom A plays the role of a branching variable of a SAT solver. And indeed, like for SAT solvers, the selection of a "good" atom A is crucial for the performance of an ASP system. In the next section, we describe a number of heuristic criteria for the selection of such branching atoms.

Remark 1. On hard ASP programs (non-hcf programs), a very large part of the computation-time may be consumed by function isAnswerSet(I)*, since it performs a co-NP-complete task if the program is non-hcf.* ∎

[3] Note that $ground(\mathcal{P})$ is not the full set of all syntactically constructible instances of rules in $\mathcal{P}$; rather, it is a subset of it having precisely the same answer sets as $\mathcal{P}$.

[4] Observe that the interpretations built during the computation are 3-valued, that is a literal can be True, False or Undefined w.r.t. to an interpretation I.s

4 Heuristics

Throughout this section, we assume that a ground ASP program $\mathcal{P}$ and an interpretation I have been fixed. Here, we describe the two heuristic criteria that will be compared in Section 5. We consider "dynamic heuristics" (the ASP equivalent of UP heuristics for SAT[5]), that is, branching rules where the heuristic value of a literal Q depends on the result of taking Q true and computing its consequences. Given a literal Q, $ext(Q)$ will denote the interpretation resulting from the application of DetCons (see previous section) on $I \cup \{Q\}$; without loss of generality, we assume that $ext(Q)$ is consistent, otherwise Q is automatically set to false and the heuristic is not evaluated on Q at all.

The Heuristic of DLV **(h_{UT}).** The heuristic employed by the DLV system was proposed in [13], where it was shown to be very effective on relevant problems like 3Satisfiability, Hamilthonian Path, Blocks World, and Strategic Companies.

A peculiar property of answer sets is *supportedness*: For each true atom A of an answer set I, there exists a rule r of the program such that the body of r is true w.r.t. I and A is the only true atom in the head of r. Since an ASP system must eventually converge to a supported interpretation, ASP systems try to keep the interpretations "as much supported as possible" during the intermediate steps of the computation. To this end, the DLV system counts the number of *UnsupportedTrue (UT)* atoms, i.e., atoms which are true in the current interpretation but still miss a supporting rule (further details on UTs can be found in [22] where they are called MBTs). For instance, the rule $:-$*not* x implies that x must be true in every answer set of the program; but it does not give a "support" for x. Thus, in the DLV system x is taken true to satisfy the rule, and it is added to the set of UnsupportedTrue; it will be removed from this set once a supporting rule for x will be found (e.g., $x \vee b{:-}c$ is a supporting rule for x in the interpretation $I = \{x, \texttt{not}\ b, c\}$). Given a literal Q, let $UT(Q)$ be the number of UT atoms in $ext(Q)$. Moreover, let $UT_2(Q)$ and $UT_3(Q)$ be the number of UT atoms occurring, respectively, in the heads of exactly 2 and 3 unsatisfied rules w.r.t. $ext(Q)$. The heuristic h_{UT} of DLV considers $UT(Q)$, $UT_2(Q)$ and $UT_3(Q)$ in a prioritized way, to favor atoms yielding interpretations with fewer $UT/UT_2/UT_3$ atoms (which should more likely lead to a supported model). If all UT counters are equal, then the heuristic considers the total number $Sat(Q)$ of rules which are satisfied w.r.t. $ext(Q)$.

The heuristic h_{UT} is "balanced", that is, the heuristic values of an atom Q depends on both the effect of taking Q and `not` Q.

For an atom Q, let $UT'(Q) = UT(Q) + UT(\texttt{not}\ Q)$, $UT_2'(Q) = UT_2(Q) + UT_2(\texttt{not}\ Q)$, $UT_3'(Q) = UT_3(Q) + UT_3(\texttt{not}\ Q)$, and, eventually, $Sat'(Q) = Sat(Q) + Sat(\texttt{not}\ Q)$. Given two atoms A and B:

1. $A <_{h_{UT}} B$ if $UT'(A) > UT'(B)$;
2. otherwise, $A <_{h_{UT}} B$ if $UT'(A) = UT'(B)$ and $UT_2'(A) > UT_2'(B)$;

[5] The UP heuristic for SAT adds for each variable x a unit clause x and -x, respectively, and makes two independent unit propagations. The choice is then based on information thus obtained.

3. otherwise, $A <_{h_{UT}} B$ if $UT'_2(A) = UT'_2(B)$ and $UT'_3(A) > UT'_3(B)$;
4. otherwise, $A <_{h_{UT}} B$ if $UT'_3(A) = UT'_3(B)$ and $Sat'(A) < Sat'(B)$.

A $<_{h_{UT}}$-maximum atom A is selected by the heuristic h_{UT} of DLV; A is taken positive or negative, by comparing the values of $UT(A)$, $UT_2(A)$, $UT_3(A)$, and $Sat(A)$, with $UT(\texttt{not}\ A)$, $UT_2(\texttt{not}\ A)$, $UT_3(\texttt{not}\ A)$, and $Sat(\texttt{not}\ A)$, respectively, as above.

Example 5. Consider $\mathcal{P}_6 = \{a \vee b \vee c.,\ d \vee e \vee f.,\ {:-}\texttt{not}\ w.,\ w{:-}a.,\ w{:-}d.,\ a \vee z{:-}w.,\ b \vee z{:-}w.,\ {:-}d, z.,\ {:-}a, z.,\ \}$, *and let the current interpretation* $I = \{w\}$*; atom* w *is UT.* a *and* d *are the* $<_{h_{UT}}$*-maxima, as only assuming their truth can eliminate the UT* w*. Indeed, anything apart from* a *or* d *would be a poor choice.* ∎

The New Heuristic (h_{DS}). The unsupported true atoms are, in a sense, the hardest constraints occurring in an ASP program. Indeed, as pointed out above, an unsupported true atom x is intuitively like a unary constraint ${:-}\texttt{not}\ x$, which must be satisfied. By minimizing the UT atoms and maximizing the satisfied rules, the heuristic h_{UT} tries to drive the DLV computation toward a *supported model* (i.e., all rules are satisfied and no UT exists). Intuitively, supported models have good chances to be answer sets (while unsupported models are guaranteed to be not answer sets), and, for simple classes of programs (e.g., tight stratified disjunctive programs) the supported models are precisely the answer sets. If the program is not tight and stratified, then supported models are not guaranteed to be answer sets; but answer-set checking can be done efficiently if the program is HCF.

For hard ASP programs (i.e., non-HCF programs – they express Σ_2^P-complete problems under brave reasoning), supported models are often not answer sets. Answer-set checking is computationally expensive (co-NP-complete), and may consume a large portion of the resources needed for computing an answer set.

The heuristic h_{DS}, described next, tries to drive the computation toward supported models having higher chances to be answer sets, reducing the overall number of the expensive answer-set checks. Models having a "higher degree of supportedness" are preferred, where the degree of supportedness is the average number of supporting rules for the true atoms (note that this number is higher than one, on supported models). Intuitively, if all true atoms have many supporting rules in a model M, then the elimination of an atom from the model would violate many rules, and it becomes less likely finding a subset of M which is a model of $\mathcal{P}^M$, to disprove that M is an answer set.

We next formalize this intuition to define the new heuristic h_{DS}. Given a literal Q, let $True(Q)$ be the number of true non-HCF atoms in $ext(Q)$, and let $SuppRules(Q)$ be the number of all supporting rules for non-HCF atoms w.r.t. $ext(Q)$. Intuitively, the heuristic maximizes the "degree of supportedness" of the interpretation, intended as the ratio between the number of supporting rules and the number of true atoms. Also in this case, the heuristic is "balanced", it takes into account both the atom and its complement.

Moreover, it is defined as a refinement of the heuristic h_{UT} (i.e., $A <_{h_{UT}} B \Rightarrow A <_{h_{DS}} B$). In this way, h_{DS} keeps the same nice behaviour as the well-assessed

h_{UT} on NP problems like 3SAT and Blocks-World, where h_{UT} proved to be very effective [13]; while, as we will see in Section 5 it sensibly improves on h_{UT} on hard 2QBF problems (Σ_2^P-complete). Given two atoms A and B:

1. $A <_{h_{DS}} B$ if $A <_{h_{UT}} B$;
2. otherwise, $A <_{h_{DS}} B$ if $B \not<_{h_{UT}} A$ and $DS(A) < DS(B)$

where $DS(Q) = SuppRules(Q)/True(Q) + SuppRules(\texttt{not } Q)/True(\texttt{not } Q)$.[6]

The heuristic selects a $<_{h_{DS}}$-maximum atom A; A is taken positive or negative, by comparing the degree of supportedness of A and not A.

Example 6. Reconsider Example 5 with the interpretation being $I = \{w\}$. We get $ext(a) = \{w, a, b, \texttt{not } z, \texttt{not } c\}$, $ext(d) = \{w, d, a, b, \texttt{not } z, \texttt{not } c, \texttt{not } e, \texttt{not } f\}$. $DS(a) = 3/3$, since $w \leftarrow a$; $a \vee z \leftarrow w$ and $b \vee z \leftarrow w$ are supporting rules for the three true non-HCF atoms w, a, b. On the other hand, $DS(d) = 4/3$, since $w \leftarrow d$ is an additional supporting rule for the same three true non-HCF atoms w, a, b. Therefore $a <_{h_{DS}} d$ holds. Indeed, d is a better choice than a, as it leads immediately to an answer set. a would require at least another choice, and choosing e or f would cause a failing model check. ∎

5 Comparing h_{UT} vs h_{DS}: Experiments

The proposed heuristic aims at improving the performance of DLV on hard (Σ_2^P-complete) ASP programs. While there are many experimental works benchmarking ASP systems on NP-complete problems, less is available for Σ_2^P-complete problems. We resort to 2QBF, the canonical problem, and one of the few Σ_2^P-hard problems for which some transition phase results are known [14,15].

The problem here is to decide whether a quantified Boolean formula (QBF) $\Phi = \exists X \forall Y \phi$, where X and Y are disjoint sets of propositional variables and $\phi = C_1 \vee \ldots \vee C_k$ is a 3DNF formula over $X \cup Y$, is valid. The transformation from 2QBF to disjunctive logic programming is based on a reduction used in [23]. The propositional disjunctive logic program $\mathcal{P}_\phi$ produced by the transformation requires $2 * (|X| + |Y|) + 1$ predicates (with one dedicated predicate w).

Our benchmark instances were generated following recent works presented in the literature that describe transition phase results for QBFs [14,15], see [9], for a thorough discussion. In all generated instances, the number of $\forall$-variables in any formula is the same as the number of $\exists$-variables (that is, $|X| = |Y|$) and each disjunct contains at least two universal variables. Moreover, the number of clauses is $((|X| + |Y|)/2)^{0.5}$.

Experiments were performed on a PentiumIV 1500 MHz machine with 256MB RAM running SuSe Linux 9.0. Time measurements have been done using the `time` command shipped with SuSe Linux 9.0.

We generated 100 random QBF instances for each problem size. The results of our experiments are displayed in Fig. 3. For each instance, we allowed a maximum time of 7200 seconds (two hours). The line of a system stops whenever

[6] The denominator is increased by 1, in order to avoid possible divisions by zero.

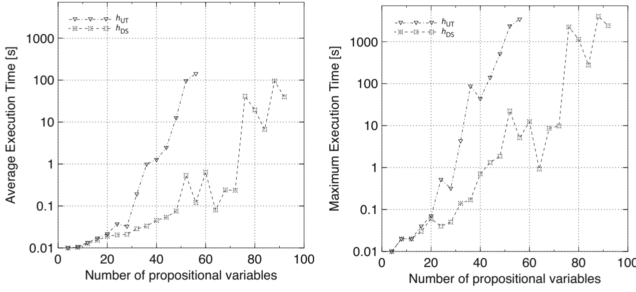


Fig. 3. Running Times on Random QBF problems

some problem instance was not solved within this time limit. On the vertical axis, we report, respectively, the average and the maximum running time in seconds over the 100 instances of the respective size, in logarithmic scale.

It is evident that the new heuristic h_{DS} outperforms the heuristic h_{UT} in these experiments. Heuristic h_{UT} stopped at size 56; while heuristic h_{DS} solved all instances up to size 92. To solve an instance of size 56, h_{UT} took 3455.85s; while h_{DS} required at most 5.13s and 0.12s on average for instances of this size. Heuristic h_{UT} could not solve a 60-variables instance within 2 hours of cpu time; while h_{DS} took at most 12.41s and 0.64s on average for solving these.

6 ASP vs QBF Solvers

One may wonder whether ASP systems are competitive with other systems on Σ_2^P/Π_2^P-hard problems. Currently it seems that QBF solvers are the most prominent (and efficient) non-ASP-systems for such problems.

In order to answer this question, we carry out an experimental comparison of DLV (with the heuristic described in this paper) with QBF solvers which performed best at the 2004 QBF evaluation [16,17]: SSolve [24] (in the version used at the 2004 QBF evaluation), Semprop [25] (version v01.06.04), Quantor [26] (version 1.3pre1), and yQuaffle [27] (version 093004). We use two different sets of benchmarks, which we describe in the following sections.

6.1 QBF Evaluation

The first group of benchmarks constitute the Σ_2^P- and Π_2^P-complete QBF instances of the 2004 QBF evaluation, which we obtained from the qbflib web site [16]. These instances are of four different kinds: (1) Letz-tree, (2) Narizzano-robot, (3) Pan-Kph, and (4) hard random-instances, see [16] for details. In total, our suite contains 143 2QBF instances: 2 Letz-tree, 32 Narizzano-robot, 1 Pan-Kph, and 108 random instances. For DLV we used a standard propositional encoding as described in Sec. 5, while for the QBF systems we used directly the qDimacs format.

The experiments were performed on the same machine as those of Sec. 5. For each instance, we have allowed a maximum running time of 1800 seconds (30 minutes). Again, we have limited the process size to 256MB to avoid swapping.

Table 1. Number (and percentage) of instances solved within the allowed time

	DLV	Quantor	Semprop	yQuaffle	SSolve
Robot	32 (100%)	10 (31%)	17 (53%)	21 (67%)	22 (69%)
Random	108 (100%)	14 (13%)	96 (89%)	55 (51%)	103 (95%)
Tree	2 (100%)	2 (100%)	2 (100%)	2 (100%)	2 (100%)
$Pan - Kph$	1 (100%)	1 (100%)	1 (100%)	1 (100%)	1 (100%)
Total	143 (100%)	27 (19%)	116 (81%)	79 (55%)	128 (90%)

Table 1 displays, for each system, the number and percentage of instances which have been solved under the resource limitations. Summarizing, DLV could solve all instances (100%) and is therefore clearly the best among the compared systems. Among the QBF solvers, SSolve and Semprop could solve 81% and 88% of the instances, respectively, and thus performed significantly better than both yQuaffle (55%) and Quantor (19%). It should be noted that practically all of the unsolved instances for Quantor are due to excessive memory consumption, while for the other systems they are due to time-outs. Indeed, we have tried to run Quantor on some of its unsolved instances manually: Within the first minute of CPU time (several minutes real-time due to swapping), it had typically allocated around 500MB, and after two minutes (around half an hour in real time) more than 700MB, still growing. We then aborted the test to avoid a machine lock-up.

Table 2. Average time (seconds) on instances solved by QBF systems

	SSolve	Semprop	Quantor	yQuaffle
# solved	128	116	27	79
solver avg	43,86	68,18	4,74	55,24
DLV avg	38,95	43,50	10,94	49,05

While SSolve and Semprop did significantly better on the random instances than on the "Narizzano-robot" instances, the situation is inverse for Quantor and yQuaffle, which confirms the observations in [17].

Also when comparing the average runtime between DLV and each QBF solver (on the instances solved by the respective system), DLV usually has an edge, as Table 2 shows. The average runtime of DLV is only larger when comparing to Quantor; but given that this comparison is based only on 19% of all instances, this is rather insignificant.

6.2 Strategic Companies

The second group of benchmarks is made up of randomly generated instances for the *Strategic Companies* problem, as defined in [28]. We use the same DLV program and generation method as in [9]. s

Here, we generated tests as in [9] with 20 instances each size for m companies ($5 \leq m \leq 200$), $3m$ products, 10 uniform randomly chosen *contr_by* relations per company (up to four controlling companies), and uniform randomly chosen

prod_by relations (up to four producers per product). The problem is deciding whether two fixed companies (1 and 2, without loss of generality) are strategic.

For the QBF solvers we have produced the following formula: $\exists c_1, \ldots, c_n : \forall c'_1, \ldots, c'_n : ((I \wedge NE) \rightarrow (R \wedge R') \wedge c_1 \wedge c_2)$ where I stands for $(c'_1 \rightarrow c_1) \wedge \ldots \wedge (c'_n \rightarrow c_n)$, NE for $\neg((c'_1 \leftrightarrow c_1) \wedge \ldots \wedge (c'_n \leftrightarrow c_n))$, R for $\bigwedge_{i=1}^{m}((\bigwedge_{c_j \in O_i} c_j) \rightarrow c_i) \wedge \bigwedge_{i=1}^{n}(\bigvee_{g_i \in C_j} c_j)$ (O_i contains the controlling companies of c_i, while C_j contains the companies producing good j. R' is defined analogous to R on the primed variables.

Unfortunately this formula is not in CNF, as required by the qDimacs format. In order to avoid a substantial blowup of the formula by a trivial normalization, we have used the tool *qst* of the *traquasto* suite [29], which transforms a formula into qDimacs by introducing additional "label variables" to avoid exponential formula growth. However, these additional variables are existentially quantified at the inner level and thus would turn the formula above into a 3QBF. To avoid this, we consider the negated formula $\forall c_1, \ldots, c_n : \exists c'_1, \ldots, c'_n : \neg((I \wedge NE) \rightarrow (R \wedge R') \wedge c_1 \wedge c_2)$, which stays on the second level after the transformation.

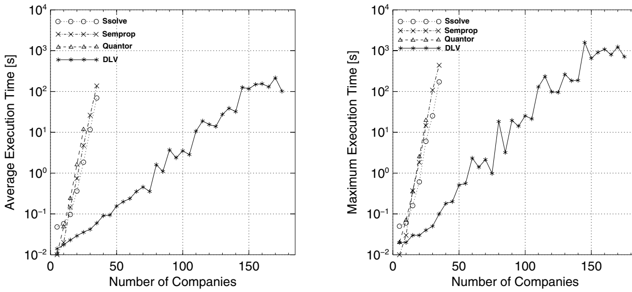


Fig. 4. Average (left) and maximum (right) timings for Strategic Companies

In the same experimental setting as before, we obtained the results of Fig. 4.[7] It is evident that DLV scales significantly better than the QBF solvers (note that the vertical axis is logarithmic), and can solve all instances of up to 175 companies, while the QBF solver fail to solve instances of 40 companies.

7 Conclusion

In this paper, we have presented a new heuristic method for ASP systems, which is geared towards hard problems on the second level of the polynomial hierarchy. We have implemented this method in the state-of-the-art system DLV, and showed that it is beneficial for the performance of the system.

To our knowledge, this is the first work dealing with heuristics dedicated for Σ_2^P/Π_2^P-hard ASP programs. Previous optimization techniques for this segment have been concerned with the model checking portion, which is important for this class of problems. In our work, we attack the problem earlier, in the model

[7] yQuaffle is not included, as it triggered assertions on some of the input files.

generation phase, and can therefore cut on the model checks. Importantly, this heuristics has been incorporated in a way such that the benign behavior on NP/co-NP programs w.r.t. the previous heuristic of DLV is maintained.

We experimentally verified that the new heuristic significantly improves the DLV system performance on randomly generated hard 2QBF instances, reducing the average execution time, enlarging the maximum solvable size of these problems for a fixed time limit.

We also carried out an experimental comparison of DLV (with the heuristic described in this paper) with the best QBF solvers of the 2004 QBF evaluation [16,17]: SSolve [24], Semprop [25], Quantor [26], and yQuaffle [27]. This comparison was done on benchmark instances of the 2004 QBF evaluation, and Strategic Companies. In both cases, DLV could outperform the QBF solvers, often significantly. DLV was able to solve *all* the instances of the 2004 QBF evaluation within the given resource limitations, while the best QBF system solved 88%, and the worst only 19%. Also for Strategic Companies, DLV exhibited much better performance. We therefore conclude that ASP systems are currently the best choice for solving Σ_2^P/Π_2^P-complete problems. All benchmark data is available at `http://www.dlvsystem.com/examples/tests-sigma2-2005.tar.gz`.

We note again that QBF solvers are designed for solving also harder problems than the ones considered here. Nevertheless, they are used for solving problems of this kind, especially planning problems, cf. [30]. However, from our results we have to conclude that DLV appears to be the better choice for Σ_2^P/Π_2^P-complete problems

References

1. Lifschitz, V.: Answer Set Planning. In Schreye, D.D., ed.: ICLP'99, Las Cruces, New Mexico, USA, The MIT Press (1999) 23–37
2. Eiter, T., Gottlob, G., Mannila, H.: Disjunctive Datalog. ACM TODS **22** (1997) 364–418
3. Rintanen, J.: Improvements to the Evaluation of Quantified Boolean Formulae. In Dean, T., ed.: IJCAI 1999, Sweden,(1999) 1192–1197
4. Eiter, T., Gottlob, G.: The Complexity of Logic-Based Abduction. JACM **42** (1995) 3–42
5. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. CUP (2002)
6. Leone, N., Rosati, R., Scarcello, F.: Enhancing Answer Set Planning. In: IJCAI-01 Workshop on Planning under Uncertainty and Incomplete Information. (2001) 33–42
7. Janhunen, T., Niemelä, I., Simons, P., You, J.H.: Partiality and Disjunctions in Stable Model Semantics. In: KR 2000, 12-15,(2000) 411–419
8. Koch, C., Leone, N., Pfeifer, G.: Enhancing Disjunctive Logic Programming Systems by SAT Checkers. Artificial Intelligence **15** (2003) 177–212
9. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV System for Knowledge Representation and Reasoning. ACM TOCL (2005) To appear.

10. Simons, P., Niemelä, I., Soininen, T.: Extending and Implementing the Stable Model Semantics. Artificial Intelligence **138** (2002) 181–234
11. Lin, F., Zhao, Y.: ASSAT: Computing Answer Sets of a Logic Program by SAT Solvers. In: AAAI-2002, Edmonton, Alberta, Canada, AAAI Press / MIT Press (2002)
12. Lierler, Y., Maratea, M.: Cmodels-2: SAT-based Answer Set Solver Enhanced to Non-tight Programs. In: LPNMR-7. LNCS, (2004) 346–350
13. Faber, W., Leone, N., Pfeifer, G.: Experimenting with Heuristics for Answer Set Programming. In: IJCAI 2001, Seattle, WA, USA,(2001) 635–640
14. Cadoli, M., Giovanardi, A., Schaerf, M.: Experimental Analysis of the Computational Cost of Evaluating Quantified Boolean Formulae. In: AI*IA 97. Italy, (1997) 207–218
15. Gent, I., Walsh, T.: The QSAT Phase Transition. In: AAAI. (1999)
16. Narizzano, M., Tacchella, A.: QBF Solvers Evaluation page (2002) `http://www.qbflib.org/qbfeval/index.html/`.
17. Berre, D.L., Simon, L., Tacchella, A.: Challenges in the QBF Arena: the SAT'03 Evaluation of QBF Solvers. In: SAT'03. Santa Margherita Ligure, Italy, (2003) 468–485
18. Marek, W., Subrahmanian, V.: The Relationship between Logic Program Semantics and Non-Monotonic Reasoning. In: ICLP'89, MIT Press (1989) 600–617
19. Leone, N., Rullo, P., Scarcello, F.: Disjunctive Stable Models: Unfounded Sets, Fixpoint Semantics and Computation. Information and Computation **135** (1997) 69–112
20. Baral, C., Gelfond, M.: Logic Programming and Knowledge Representation. JLP **19/20** (1994) 73–148
21. Ben-Eliyahu, R., Dechter, R.: Propositional Semantics for Disjunctive Logic Programs. AMAI **12** (1994) 53–87
22. Faber, W., Leone, N., Pfeifer, G.: Pushing Goal Derivation in DLP Computations. In: LPNMR'99. LNCS 1730
23. Eiter, T., Gottlob, G.: On the Computational Cost of Disjunctive Logic Programming: Propositional Case. AMAI **15** (1995) 289–323
24. Feldmann, R., Monien, B., Schamberger, S.: A Distributed Algorithm to Evaluate Quantified Boolean Formulae. In: Proceedings National Conference on AI (AAAI'00), Austin, Texas, AAAI Press (2000) 285–290
25. Letz, R.: Lemma and Model Caching in Decision Procedures for Quantified Boolean Formulas. In: TABLEAUX 2002. Denmark, (2002) 160–175
26. Biere, A.: Resolve and Expand. (2004) SAT'04.
27. Zhang, L., Malik, S.: Towards a Symmetric Treatment of Satisfaction and Conflicts in Quantified Boolean Formula Evaluation. In: CP 2002. NY, USA, (2002) 200–215
28. Cadoli, M., Eiter, T., Gottlob, G.: Default Logic as a Query Language. IEEE TKDE **9** (1997) 448–463
29. Zolda, M.: Comparing Different Prenexing Strategies for Quantified Boolean Formulas. Master's thesis, TU Wien (2005)
30. Castellini, C., Giunchiglia, E., Tacchella, A.: SAT-based planning in complex domains: Concurrency, constraints and nondeterminism. Artificial Intelligence **147** (2003) 85–117

Mode-Directed Fixed Point Computation

Hai-Feng Guo

Department of Computer Science,
University of Nebraska at Omaha, Omaha, NE 68182-0500, USA
haifengguo@mail.unomaha.edu

Abstract. Goal-directed fixed point computation strategies have been widely adopted in the tabled logic programming paradigm. However, there are many situations in which a fixed point contains a large number or even infinite number of solutions. In these cases, a fixed point computation engine may not be efficient enough or feasible at all. We present a mode-declaration scheme which provides the capabilities to reduce a fixed point from a big solution set to a preferred small one, or from an infeasible infinite set to a finite one. We show the correctness of the mode-declaration scheme. One motivating application of our mode-declaration scheme is for dynamic programming, which is typically used for solving optimization problems. There is no need to define the value of an optimal solution recursively, instead, defining a general solution suffices. The optimal value as well as its corresponding concrete solution can be derived implicitly and automatically using a mode-directed fixed point computation engine. This mode-directed fixed point computation engine has been successfully implemented in a commercial Prolog system.

1 Introduction

Due to their highly declarative nature and efficiency, Tabled logic programming (TLP) systems [3,16,5,10] have been put to many innovative uses, such as model checking [9] and non-monotonic reasoning [12]. A tabled logic programming system can be thought of as an engine for efficiently computing fixed points, which is critical for many practical applications. A TLP system is essential for extending traditional LP system (e.g., Prolog) with tabled resolutions (or memorized resolutions). The main advantages of tabled resolution are that a TLP system terminates more often by computing fixed points, avoids redundant computation by memoing the computed answers, and keeps the declarative and procedural semantics consistent for pure logic programs with bounded-size terms.

Example 1. *Consider the following two programs defining the reachability relations. The predicate* `reach(X,Y)` *in the program (a) checks whether a node* `X` *is reachable to* `Y`*, while the predicate* `reach(X,Y,E)` *in the program (b) does the same job as well as returning a path information as an explanation.*

C. Baral et al. (Eds.): LPNMR 2005, LNAI 3662, pp. 253–265, 2005.
© Springer-Verlag Berlin Heidelberg 2005

```
:- table reach/2.
reach(X,Y) :- reach(X,Z), arc(Z,Y).
reach(X,Y) :- arc(X,Y).
arc(a,b). arc(a,c). arc(b,a).
:- reach(a,X).
```

(a) Finite Solutions

```
:- table reach/3.
reach(X,Y,E) :-
 reach(X,Z,E1), arc(Z,Y,E2),
 append(E1,E2,E).
reach(X,Y,E) :- arc(X,Y,E).
arc(a,b,[(a,b)]). arc(a,c,[(a,c)]).
arc(b,a,[(b,a)]).
:- reach(a,X,P).
```

(b) Infinite Solutions

The program for example 1(a), checking the existence of reachability, does not work properly in a traditional Prolog system. With the declaration of a tabled predicate `reach/2` in a tabled Prolog system, it can successfully find the complete solutions due to the fixed point computation strategy. However, there are many situations in which a fixed point contains a large number or even infinite number of solutions, which in turn affects the efficiency or completion of the computation.

Consider another reachability program shown in example 1(b), where `append/3` is a standard predicate to append a list to another. An extra argument is added for the predicate `reach/3` to collect the corresponding path. However, this extra argument results in the fixed point of the computation to be infinite and nonterminating, since there are indeed infinite number of paths from `a` to any node due to the cycle between `a` and `b`. Similar problems on evidence construction have been studied on justification in [11,8]. One reasonable solution is presented in [8] by asserting the first evidence into a dynamic database for each tabled answer. However, the evidence has to be organized as segments indexed by each tabled answer. That is, an extra procedure is required to construct the full evidence.

To avoid such an inefficiency or nontermination problem due to a fixed point containing a large number or an infinite number of solutions respectively, it is often necessary to change the original problem to having a small finite solution set. For this reachability example, it is actually enough to find a single simple path to show the evidence of reachability. However, it is generally not only difficult to alter the predicate definition of `reach/3` to avoid nontermination to have a single simple path for each pair of reachable nodes, but also sacrifices the clarity of the original relation. In these cases, a fixed point computation engine may not be efficient enough or feasible at all.

In this paper, we present a mode-declaration scheme in a tabled Prolog paradigm which provides the capabilities to reduce a fixed point from a large or infinite solution set to a preferred finite one. Note that we only consider definite logic programs in this paper. The method introduces a new mode declaration for tabled predicates. The mode declaration classifies arguments of a tabled predicate as indexed or non-indexed. Each non-indexed argument can be thought of as a function value uniquely determined by indexed arguments. The mode declaration can further extend one of the non-indexed arguments to be an aggregated value, e.g., the minimum function, so that the global *table* will record answers

with the value of that argument appropriately aggregated. Thus, in the case of the minimum function, a tabled answer can be dynamically replaced by a new one with a smaller value during the computation.

Semantically, the mode declaration scheme can be characterized as a meta-level operation over the fixed point to the original program. The semantics of a tabled Prolog program is formalized based on the Herbrand model [13,7] and fixed-point theory, whereas the semantics of declared modes is defined as a strict partial order relation [1] among the solutions. The mode declaration essentially provides selection mechanism among the alternative solutions, thus making fixed point computation more flexible. We formally present the semantics of mode declaration in a tabled Prolog program, and further show the correctness of its operational semantics in a tabled resolution.

The new mode-declaration scheme, coupled with recursion, provides an attractive platform for making dynamic programming simpler [6]: there is no need to define the value of an optimal solution recursively, instead, defining the value of a general solution suffices. The optimal value, as well as its associated solution, will be computed implicitly and automatically in a tabled Prolog system that uses the new mode declaration and modified variant checking. Thus, dynamic programming problems are solved more elegantly and declaratively.

The rest of the paper is organized as follows: Section 2 gives a brief introduction of tabled logic programming and presents a mode declaration scheme for tabled predicates. Section 3 explains how the mode declaration affects the operational semantics of tabled logic programming. Section 4 gives a detailed demonstration of how dynamic programming can benefit from this new scheme. Section 5 presents the running performance on some dynamic programming benchmarks. Finally, section 6 gives our conclusions.

2 Mode-Directed Fixed Point Computation

2.1 Tabled Logic Programming (TLP)

Traditional logic programming systems (e.g., Prolog) use SLD resolution [7] with the following *computation strategy*: subgoals of a resolvent are solved from left to right and clauses that match a subgoal are applied in the textual order they appear in the program. It is well known that SLD resolution may lead to non-termination for certain programs, even though an answer may exist via the declarative semantics. That is, given any static computation strategy, one can always produce a program in which no answers can be found due to non-termination even though some answers may logically follow from the program. In case of Prolog, programs containing certain types of left-recursive clauses are examples of such programs.

Tabled logic programming (TLP) [3,5,16,10] eliminates such infinite loops by extending logic programming with tabled resolution. The main idea of tabled resolution is to memorize the answers to some calls and use the memorized

[1] A strict partial order relation is irreflexive and transitive.

answers to resolve subsequent variant calls. Tabled resolution adopts a dynamic computation strategy while resolving subgoals in the current resolvent against matched program clauses or tabled answers. It keeps track of the nature and type of the subgoals; if the subgoal in the current resolvent is a variant of a former tabled call, tabled answers are used to resolve the subgoal; otherwise, program clauses are used following SLD resolution.

TLP systems can be thought of as an engine for efficiently computing fixed points. However, there are many situations in which a fixed point contains a large number or even infinite number of solutions (for example 1(b)). In these cases, a fixed point computation engine may not be efficient enough or feasible at all. In this section, we present a mode-declaration scheme which provides the capabilities to reduce a fixed point from a big solution set to a preferred small one, or from an infeasible infinite set to a finite one.

2.2 Mode Declarations

The fixed point reduction can be achieved by a mode declaration for tabled predicates, which is described in the form of

`:- table` $q(m_1, ..., m_n)$`.`

where q/n is a tabled predicate name, $n \geq 0$, and each m_i has one of the forms as defined in Table 1.

Table 1. Built-in Modes for Tabled Predicates

Modes	Informal Semantics
$+$	an indexed argument
$-$	a non-indexed argument
min	a minimum non-indexed argument
max	a maximum non-indexed argument

In order to find out how modes will affect the fixed point computation, we have to get better understanding on the function of variant checking in tabled resolution. Variant checking is a crucial operation for tabled resolution as it leads to avoidance of non-termination. It is used to differentiate both tabled goals and their answers. While computing the answers to a tabled goal p with tabled resolution, if another tabled subgoal q is encountered, the decision regarding whether to consume tabled answers or to try program clauses depends on the result of variant checking. If q is a variant of p, the variant subgoal q will be resolved by unifying it with tabled answers, otherwise, traditional Prolog resolution is adopted for q. Additionally, when an answer to a tabled goal is generated, variant checking is used to check whether the generated answer is variant of an answer that is already recorded in the table. If so, the table is not changed; this step is crucial in ensuring that a fixed point is reached.

The main purpose of the mode declaration is to classify the predicate arguments into two types: *indexed* and *non-indexed*. Only indexed arguments are

used for variant checking during collecting answers for the table; for each tabled call, any answer generated later for the same value of the indexed arguments is discarded because it is a variant, w.r.t. the indexed arguments, of a previously tabled answer. Consider again the reachability program in Example 1(b). Suppose we declare the mode as "`:- table reach(+,+,-)`"; this means that only the first two arguments of the predicate `reach/3` are used for variant checking during the tabled answers collection. Given the query `reach(a,X,P)`, the answers "`X=b, P=[(a,b),(b,a),(a,b)]`" and "`X=c, P=[(a,b),(b,a),(a,c)]`", are variant to "`X=b, P=[(a,b)]`" and "`X=c, P=[(a,c)]`" respectively. Therefore, the computation is then terminated properly with three answers, that is, each answer gives a reachable node from `a` as well as its explanation.

The mode directive `table` makes it very easy and efficient to extract explanation for tabled predicates. In fact, our strategy of ignoring the explanation argument during variant checking results in only the first explanation for each tabled answer being recorded. Subsequent explanations are filtered by our modified variant checking scheme. This feature ensures that those generated explanations are concise and that cyclic explanations are guaranteed to be absent. For the reachability instance, each returned path is simple such that all arcs are distinct.

Essentially, if we regard a tabled predicate as a function, then all the non-indexed arguments are uniquely defined by the instances of indexed arguments. For the previous example, the third argument of `reach/3` returns a single path depending on the first two arguments. Therefore, variant checking should be done w.r.t. only indexed arguments during tabled resolution. From this viewpoint, the mode declaration makes tabled resolution more efficient and flexible. More importantly, this declaration scheme is especially useful to reduce an infinite computation model to a finite one for some practical uses, or to reduce a large finite computation model to an optimized one as shown below.

2.3 Declaration of Aggregates

The mode directive `table` can be further extended to associate a non-indexed argument of a tabled predicate with some optimum constraint. With the mode '`-`', a non-indexed argument for each tabled answer only records the very first instance. This "very first" property can actually be generalized to support other preferences, e.g., the minimum value with mode `min` (or the maximum with mode `max`), in which case the global table will record answers with the value of that argument as small (or great) as possible. That is, a tabled answer can be dynamically replaced by a new one with smaller (or greater) value during the computation. Note that we only enumerate two typical aggregates as examples, other aggregates, such as *sum*, *average*, etc., can easily be extended as well.

Example 2. *Consider the following program searching for a shortest path, where* `path(X,Y,D,L)` *denotes a path from* `X` *to* `Y` *with the distance* `D` *and the route* `L`.

```
:- table path(+, +, min, -).                                    (1)
path(X, X, 0, []).                                              (2)
path(X, Y, D, [e(X, Y)]) :- edge(X, Y, D).                      (3)
path(X, Y, D, [e(X, Z) | P]) :-
  edge(X, Z, D1), path(Z, Y, D2, P), D is D1 + D2.              (4)
edge(a,b,4).    edge(b,a,3).    edge(b,c,2).                    (5)
:- path(a, X, D, P).                                            (6)
```

The aggregates, `min` and `max`, are specified via mode declarations as shown in Table 1. Both modes imply that the declared arguments are non-indexed. The aggregates can be used to make the specification and execution of optimization problems more elegant. For the program 2 searching for shortest paths, instead of defining the shortest path directly, we only need to specify what is the definition for a general path. Clauses (2) to (5) make up the core program defining the path relation and a directed graph with a set of edges; Clause (1) specifies the predicate `path/4` to be optimized and gives the criteria how to optimize the `path/4` predicate. The mode declaration `path(+,+,min,-)` means that only the first two arguments (pair of nodes) are used for variant checking when an answer is generated, and a minimum value (the shortest path) is expected for the third argument. Arguments with different modes are tested in the following order during variant checking of a recently generated answer: (i) the indexed argument with '`+`' mode has the highest priority to be checked to identify whether it is a new answer. If that is the case, a new tabled entry is required to record the answer; otherwise a tabled answer with the same indexed arguments is found. (ii) This tabled answer is then compared with the recently generated one w.r.t the argument with the optimum mode '`min`'; if the new answer has a smaller value on the optimum argument, then a replacement of the tabled answer is required such that the tabled answer keeps the minimum value as expected for this argument. (iii) The last argument with mode '`-`' will not be used for variant checking; if a replacement of a tabled answer happens, then the argument will be replaced as well; otherwise, the recently generated answer as well as its fourth argument are discarded.

3 Operational Semantics

The operational semantics of a tabled program is dependent on tabled resolution [2,16,5], which can be formalized based on the Herbrand model [13,7] and fixed-point theory. In spite of having different tabled resolution, a tabled Prolog can be thought of as an engine for computing the least fixed points by mimicing the bottom-up computation strategy [5]. For the consideration of clarity and simplicity, we ignore any optimization used for the bottom-up computation. (e.g. incremental consumption simulating semi-naive bottom-up computation).

We use the following notational conventions: P is used to denote a tabled logic program, B_P denotes the Herbrand base of P, 2^{B^P} denotes the set of all Herbrand interpretations of P, and a ground instance (e.g., a ground atom, a ground instance of a clause) denotes an instance without involving any variable.

Note that ω is the first infinite ordinal, and $\mathcal{F} \uparrow n(x)$ to denote applying the mapping $\mathcal{F}$ n times as $\overbrace{\mathcal{F}(\mathcal{F}(\cdots\mathcal{F}}^{n}(x)\cdots))$.

Definition 1. *Let P and B_P be a logic program and its Herbrand base. We define a meta-level procedure $T_P : 2^{B_P} \rightarrow 2^{B_P}$. Given a Herbrand interpretation I, $T_P(I)$ performs:*

1. $I_0 \leftarrow \emptyset$;

2. **for** *each ground instance A :- $A_1, \cdots, A_n$ of a clause in P where $\{A_1, \cdots, A_n\} \subseteq I$,* **do**

  $I_0 \leftarrow I_0 \cup \{A\}$;

3. **return** I_0.

Thus, the fixed point semantics of P can be described as $T_P \uparrow \omega(\emptyset)$ [7].

We next show how mode declaration affects the fixed point semantics of a logic program. One key ingredient that the mode declaration scheme can be applicable is the *optimal-substructure* property [2], that is, the optimal solution to a tabled call contains optimal solutions to its tabled sub-calls. Typical examples of such problems are those for dynamic programming. For simplicity, we assume that for any tabled predicate, there is at most one optimization mode, '`min`' or '`max`', in the mode declaration. For the optimization minimizing or maximizing multiple arguments, they can always be combined as one by transforming the program clauses. Additionally, we assign non-indexed modes different priorities, i.e., '`min`' and '`max`' have higher priorities than '`-`'.

Note that a tabled predicate without explicit mode declaration has a default one with all indexed modes '`+`'. Although mode declarations are only allowed for tabled predicates, the non-tabled predicates can also be simply treated as implicitly declaring indexed modes '`+`' for all arguments. Thus, in the rest of this subsection we will not differentiate tabled predicates from non-tabled predicates, and each predicate defined in a tabled logic program is associated with a mode declaration.

Definition 2. *Let q/n be a predicate with a mode declaration $q(m_1, m_2, \cdots, m_n)$ in a tabled logic program P, let $m_{i1}, m_{i2}, \cdots, m_{ik}$ $(0 \leq k \leq n)$ be all the modes '`+`' such that $1 \leq i1 < i2 < \cdots < ik \leq n$; let m_j be a non-indexed mode with the highest priority if there are any non-indexed modes for q/n; we define two functions $\mathcal{K}_{q/n}$ and $\mathcal{O}_{q/n}$ as follows: given a ground atom $q(a_1, a_2, \cdots, a_n)$,*

$$\mathcal{K}_{q/n}(q(a_1, a_2, \cdots, a_n)) = (a_{i1}, a_{i2}, \cdots, a_{ik});$$
$$\mathcal{O}_{q/n}(q(a_1, a_2, \cdots, a_n)) = a_j, \quad \text{if } m_j \text{ exists.}$$

We say two ground atoms, t_1 and t_2, of q/n comparable *if and only if $\mathcal{K}_{q/n}(t_1) = \mathcal{K}_{q/n}(t_2)$.*

[2] The optimal-substructure property refers to problems with optimal solutions that exhibit optimal solutions in their subproblems.

The function $\mathcal{K}_{q/n}$ is used to return a sequence of indexed arguments in a left-to-right order, whereas $\mathcal{O}_{q/n}$ return a non-indexed argument with the highest priority. We say two ground atoms of q/n *comparable* if and only if these two atoms have the same indexed arguments. We abbreviate $\mathcal{K}_{q/n}$ and $\mathcal{O}_{q/n}$ to $\mathcal{K}$ and $\mathcal{O}$, respectively, whenever the predicate is obvious from the context. Thus, we have a preference relation defined as follows.

Definition 3. *Let P be a logic program. A* preference relation *in P is a strict partial order relation $\prec_P$ s.t. for any two ground atoms A_1 and A_2 of a predicate q/n in P, $A_1 \prec_P A_2$ if both of the followings are true:*

- $\mathcal{K}(A_1) = \mathcal{K}(A_2)$;
- **cases** *the non-indexed mode with the highest priority in q/n* **of**
  min: $\mathcal{O}(A_1) is comparably greater than \mathcal{O}(A_2)$;
  max: $\mathcal{O}(A_1) is comparably less than \mathcal{O}(A_2)$;
  $-$: *A_2 is generated earlier than A_1 during tabled resolution.*

We abbreviate $\prec_P$ to $\prec$ whenever the tabled logic program is obvious from the context. It has to be mentioned that the semantics of mode '$-$' is heavily dependent on the generating order of answers, which is decided by the procedure of tabled resolution. For Example 2, the preference relation $\prec$ is the set

$$\begin{array}{l}\{\ \text{path}(a,a,7,_) \prec \text{path}(a,a,0,_), \text{path}(a,a,14,_) \prec \text{path}(a,a,0,_), \quad \cdots \\ \text{path}(a,b,11,_) \prec \text{path}(a,b,4,_), \text{path}(a,b,18,_) \prec \text{path}(a,b,4,_), \quad \cdots\ \cdots\}\end{array}$$

where the numbers $0, 7, 11, ...$ are the possible distances for their corresponding pair of nodes, and '_' means any ground term from the Herbrand universe. Note that no atoms of the non-tabled predicate `edge/3` are in the preference relation. That is because the non-tabled predicate has an implicit declaration having indexed modes '+' for all arguments, therefore, none of their atoms can satisfy the second condition of Definition 3. Similarly, all ground atoms of non-tabled predicates are *optimized* according to the following definition.

Definition 4. *Let P be a logic program and I be one of its Herbrand interpretations; We say that A is* an optimized ground atom, *abbreviated as* an optimized atom, *in I if there does not exist any other ground atom $A_1 \in I$ s.t. $A \prec A_1$.*

Definition 5. *Let P and B_P be a logic program and its Herbrand base. We define a meta-level procedure $T_P^{'} : 2^{B_P} \rightarrow 2^{B_P}$. Given a Herbrand interpretation I, $T_P^{'}(I)$ performs:*

1. $I_0 \leftarrow \emptyset$;
2. **for** *each ground instance A :- $A_1, \cdots, A_n$ of a clause in P where* $\{A_1, \cdots, A_n\} \subseteq I$, **do**
  2a. $I_0 \leftarrow I_0 \cup \{A\}$;
  2b. $I_0 \leftarrow I_0 - \{a_1 \in I_0 : \exists a_2 \in I_0 \ s.t. \ a_1 \prec a_2\}$ (*)
3. **return** I_0.

Thus, the fixed point semantics of P can be described as $T_P^{'} \uparrow \omega(\emptyset)$.

Def. 5 gives the fixed point semantics for a tabled logic program with mode declaration. The statement (*) shows how non-indexed modes affect the the procedural semantics of the core program through the preference relation $\prec$.

Proposition 1. *Let P be a tabled logic program. For any atom $A \in T_P^{'} \uparrow n(\emptyset)$, where $n \geq 0$, A is an optimized atom in $T_P^{'} \uparrow n(\emptyset)$.*

Proof: This can be easily shown by a mathematical induction on n, mainly using the result of step $2b$ in Definition 5:

$$I_0 \leftarrow I_0 - \{a_1 \in I_0 : \exists a_2 \in I_0 \text{ s.t. } a_1 \prec a_2\},$$

so that any atom A in the resulting I_0 is an optimized atom according to Definition 4. □

Proposition 2. *Let P be a tabled logic program. If A is an optimized atom in $T_P \uparrow n(\emptyset)$, then $A \in T_P^{'} \uparrow n(\emptyset)$, for any $n \geq 0$.*

Proof: The proof is a mathematical induction on n.
`Base case:` Consider $n = 0$. Since $T_P \uparrow 0(\emptyset)$ is an empty set, the proposition is vacuously true.
`Inductive Case:` Assume that the proposition is true for some $i \geq 0$. We consider an optimized atom $A \in T_P \uparrow\ (i+1)(\emptyset)$. A is obviously an optimized atom in $T_P^{'} \uparrow (i+1)(\emptyset)$ as well due to the fact that $T_P^{'} \uparrow (i+1)(\emptyset) \subseteq T_P \uparrow\ (i+1)(\emptyset)$. Next, we complete the proof by showing $A \in T_P^{'} \uparrow (i+1)(\emptyset)$. According to Definition 1, there exists a ground instance A:-$A_1, ..., A_m$ (for some $m \geq 0$) of a clause in P where $\{A_1, ..., A_m\} \subseteq T_P \uparrow i(\emptyset)$. Based on the optimal-substructure property, $A_1, ..., A_m$ must be optimized atoms in $T_P \uparrow i(\emptyset)$. Following the induction assumption, we have $\{A_1, ..., A_m\} \in T_P^{'} \uparrow i(\emptyset)$. Therefore, A satisfies the conditions specified in Definition 5; we have $A \in T_P^{'} \uparrow (i+1)(\emptyset)$. □

Proposition 3. *Let P be a tabled logic program. If $A \in T_P^{'} \uparrow n(\emptyset)$, then A is an optimized atom in $T_P \uparrow n(\emptyset)$, for any $n \geq 0$.*

Proof: We can easily get $A \in T_P \uparrow n(\emptyset)$ due to the fact that $T_P^{'} \uparrow n(\emptyset) \subseteq T_P \uparrow n(\emptyset)$. Assume that A' is an optimized atom in $T_P \uparrow n(\emptyset)$ and $A \prec A'$. Based on Proposition 2, we have $A' \in T_P^{'} \uparrow n(\emptyset)$, which is a contradiction with the fact that $A \in T_P^{'} \uparrow n(\emptyset)$ since $A \prec A'$. Therefore, A must be an optimized atom in $T_P \uparrow n(\emptyset)$. □

Thus, we have the following major result showing the correctness of our mode declaration scheme in the application of optimization problems with optimal-substructure properties.

Theorem 4. *Let P be a tabled logic program. $A \in T_P^{'} \uparrow \omega(\emptyset)$ if and only if A is an optimized atom in $T_P \uparrow \omega(\emptyset)$.*

Proof: According to Proposition 2 and Proposition 3. □

The monotonic property in traditional logic programming is not present in the mode-directed fixed point computation. Adding new rules or facts to a logic

program may retract obtained conclusions since the new knowledge may lead to new optimal answers for mode-directed optimization predicates. Fortunately, it is not necessary to compute the new fixed point from scratch. Instead, the meta-level procedure $T_P^{'}$ in Definition 5 can continue to be applied on the obtained interpretation as well as the new knowledge, until a new fixed point is found.

4 Dynamic Programming with Modes

In the dynamic programming paradigm the value of an optimal solution is recursively defined in terms of optimal solutions to subproblems. Such dynamic programming definitions can be very tricky and error-prone to specify due to the involvement of both optimization and recursion. In this section, we presents a novel, elegant method using mode declaration that simplifies the specification of such dynamic programming solutions.

We use the *matrix-chain multiplication* problem [4] as an example to illustrate how tabled logic programming can be adopted for solving dynamic programming problems. A product of matrices is *fully parenthesized* if it is either a single matrix or the product of two fully parenthesized matrix products, surrounded by parentheses. Thus, the matrix-chain multiplication problem can be stated as follows

Problem 1. *Given a chain* $\langle A_1, A_2, ..., A_n \rangle$ *of* n *matrices, where for* $i = 1, 2, ..., n$*, matrix* A_i *has dimension* $p_{i-1} \times p_i$*, fully parenthesize the product* $A_1 A_2 ... A_n$ *in a way that minimizes the number of scalar multiplications.*

Using mode declaration for dynamic programming, the programmer is only required to define *what* a general solution is, while searching for the optimal solution is left to the TLP system. The mode declaration can be used to make control of execution implicit during dynamic programming, making the specification of dynamic programming problems more declarative and elegant. For the matrix-chain multiplication, instead of defining the cost of an optimal solution, we only need to specify what the cost for a general solution is. Let $m[i, j]$ be the number of scalar multiplications needed to compute the matrix $A_{i..j}$ for $1 \leq i \leq j \leq n$, where n is the total number of matrices. The recursive definition for the cost of parenthesizing $A_{i..j}$ becomes

$$m[i, j] = \begin{cases} 0 & \text{if } i = j, \\ m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j & \text{if } i < j, \end{cases}$$

where any $k \in [i, j)$. Thus, we have the following program shown in Example 3.

Example 3. *A tabled logic program with optimum mode declaration for matrix-chain multiplication problems:*

```
:- table scalar_cost(+, min, -, -).
scalar_cost([P1, P2], 0, P1, P2).
scalar_cost([P1, P2, P3 | Pr], V, P1, Pn) :-
```

```
break([P1, P2, P3 | Pr], PL1, PL2, Pk),
scalar_cost(PL1, V1, P1, Pk),
scalar_cost(PL2, V2, Pk, Pn),
V is V1 + V2 + P1 * Pk * Pn.
```

The predicate `scalar_cost(PL, V, P0, Pn)` is a tabled predicate, where `PL`, `P0` and `Pn` are given by the user to represent the dimension squence $[p_0, p_1, ..., p_n]$, the first dimension p_0 and the last dimension p_n, respectively, and `V` is the minimum cost of scalar multiplications to multiply $A_{1..n}$. The mode declaration `scalar_cost(+,min,-,-)` means that only the first argument (the list of matrix dimensions) is used for variant checking when an answer is generated, and a minimum value is expected from the second argument (the cost of scalar multiplication).

Consider the problem for a chain $\langle A_1, A_2, A_3 \rangle$ of three matrices. Suppose that the dimensions of the matrices are 10×100, 100×5, and 5×50, respectively. Figure 1 shows a skeleton of the recursion tree produced by the query

`:- scalar_cost([10,100,5,50],V,10,50).`

Its first tabled answer has `V=75000`. However, when the second answer `V=7500` is computed, it will automatically replace the previous answer following the declared optimum mode. Therefore, there is at most one answer for the tabled call `scalar_cost([10,100,5,50],V,10,50)` that exists in the table at any point in time, and it represents the optimal value computed up to that point.

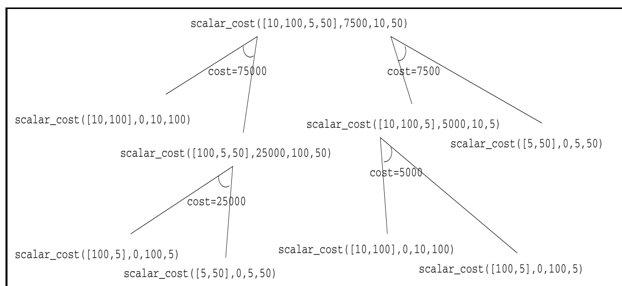


Fig. 1. Recursion tree for computing `scalar_cost([10,100,5,50],V,10,50)`

5 Experimental Results

The mode declaration scheme has been implemented in the author's TALS [5] system, a tabled Prolog system incorporating DRA resolution on the top of the commercial ALS Prolog engine [1]. Detailed implementation details and experimental results can be found in [6]. No changes are required to the DRA resolution; therefore, the same idea can also be applied to other tabled Prolog systems.

Our experimental benchmarks include five typical dynamic programming examples. `matrix` is the matrix-chain multiplication problem; `lcs` is longest

common subsequence problem; `obst` finds an optimal binary search tree; `apsp` finds the shortest paths for all pairs of nodes; and `knap` is the knapsack problem. All tests were performed in TALS system on an Intel Pentium 4 CPU 2.0GHz machine with 512M RAM running RedHat Linux 9.0.

Table 2. Running time performance comparison in Seconds(Ratio)

Benchmarks	*without modes*	*with modes*
matrix	2.74(1.0)	1.97(0.72)
lcs	0.86(1.0)	0.55(0.63)
obst	10.58(1.0)	0.63(0.06)
apsp	6.05(1.0)	2.85(0.47)
knap	126.25(1.0)	38.56(0.31)

Table 2 compares the running time performance between the programs with and without mode declaration. The experimental data indicates, based on the ratios in Table 2, that the programs with mode declaration consume only 6% to 72% time that the corresponding programs without mode declaration do.

The efficiency for those benchmarks are mainly credited to two factors. First, tabled Prolog systems with mode declaration provides a concise but easy-to-use interface for dynamic programming, and it does not introduce any major overhead; the mode declaration are flexible and powerful to support meta-level manipulation of fixed points, and the mode functionality is implemented at the system level instead of the Prolog programming level. Second, tabled answers can be more efficiently organized due to the mode declaration. Indeed, variant checking is only applied for indexed arguments; if an indexed argument is instantiated in advance before a tabled goal is called, variant checking on this indexed argument can be avoided since its value is same for all the answers; furthermore, it is not necessary to record the pre-instantiated value with each tabled answer because the same value has already been stored in the tabled call entry. Those optimization leads to great running performance improvement.

6 Conclusions

A new mode declaration for tabled predicates has been presented in TLP systems to aggregate information dynamically into the table. It provides a declarative method to reduce a fixed point from a big solution set to a preferred small one, or even from an infeasible infinite set to a finite one. The mode declaration classifies arguments of tabled predicates as either indexed or non-indexed. As a result, (i) a tabled predicate can be regarded as a function in which non-indexed arguments (outputs) are uniquely defined by the indexed arguments (inputs); (ii) concise explanation for tabled answers can be easily constructed in non-indexed (output) arguments; (iii) the efficiency of tabled resolution may be

improved since only indexed arguments are involved in variant checking; and (iv) the non-indexed arguments of a tabled predicate can be further qualified with an aggregate mode such that an optimal value can be sought without explicit coding of the comparison.

This new mode declaration scheme, coupled with recursion, provides an elegant method for specifying dynamic programming problems: there is no need to define the value of an optimal solution recursively, instead, defining the value of a general solution is enough. The optimal value, as well as its associated solution, is obtained automatically by the TLP systems. This new scheme has been implemented in the TALS system with encouraging results.

References

1. ALS. Applied logic systems, inc., http://www.als.com.
2. W. Chen and D. S. Warren. Query evaluation under the well founded semantics. In *ACM Symposium on Principles of Database Systems*, pages 168–179, 1993.
3. W. Chen and D. S. Warren. Tabled evaluation with delaying for general logic programs. *Journal of the ACM*, 43(1):20–74, January 1996.
4. T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. The MIT Press, 2001.
5. H.-F. Guo and G. Gupta. A simple scheme for implementing tabled logic programming systems based on dynamic reordering of alternatives. In *Proceedings of International Conference on Logic Programming*, pages 181–196, 2001.
6. H.-F. Guo and G. Gupta. Simplifying dynamic programming via tabling. In *Practical Aspects of Declarative Languages*, pages 163–177, 2004.
7. J. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 1987.
8. G. Pemmasani, H.-F. Guo, Y. Dong, C. Ramakrishnan, and I. Ramakrishnan. Online justification for tabled logic programs. In *International Symposium of Functional and Logic Programming*, April 2004.
9. Y. Ramakrishnan, C. Ramakrishnan, I. Ramakrishnan, S. Smolka, T. Swift, and D. Warren. Efficient model checking using tabled resolution. In *Proceedings of Computer Aided Verification*, pages 143–154, 1997.
10. R. Rocha, F. Silva, and V. S. Costa. On a tabling engine that can exploit or-parallelism. In *Proceedings of International Conference on Logic Programming*, pages 43–58, 2001.
11. A. Roychoudhury, C. Ramakrishnan, and I. Ramakrishnan. Justifying proofs using memo tables. In *Second International ACM SIGPLAN conference on Principles and Practice of Declarative Programming*, pages 178–189, 2000.
12. T. Swift. Tabling for non-monotonic programming. *Annals of Mathematics and Artificial Intelligence*, 25(3-4):201–240, 1999.
13. M. van Emden and R. Kowalski. The semantics of predicate logic as a programming language. *Journal of AMC*, 23(4):733–742, 1976.
14. D. S. Warren. *Programming in Tabled Prolog (Draft Book)*. www.cs.sunysb.edu/~warren.
15. XSB. http://xsb.sourceforge.net.
16. N.-F. Zhou, Y. Shen, L. Yuan, and J. You. Implementation of a linear tabling mechanism. In *Proceedings of Practical Aspects of Declarative Languages*, 2000.

Lookahead in Smodels Compared to Local Consistencies in CSP

Jia-Huai You, Guohua Liu, Li Yan Yuan, and Curtis Onuczko

Department of Computing Science,
University of Alberta,
Edmonton, Alberta, Canada
{you, guohua, yuan, onuczko}@cs.ualberta.ca

Abstract. In answer set programming systems like Smodels and some SAT solvers, constraint propagation is carried out by a mechanism called lookahead. The question arises as what is the pruning power of lookahead, and how such pruning power fares in comparison with the consistency techniques in solving CSPs. In this paper, we study the pruning power of lookahead by relating it to local consistencies under two different encodings from CSPs to answer set programs. This leads to an understanding of how the search space is pruned in an answer set solver with lookahead for solving CSPs. On the other hand, lookahead as a general constraint propagation mechanism provides a uniform algorithm for enforcing a variety of local consistencies. We also study the impact on the search efficiency under these encodings.

1 Introduction

Constraint satisfaction problems (CSPs) on the one hand and propositional satisfiability (SAT) and answer set programming (ASP) under the stable model semantics [9] on the other are two competing approaches to constraint programming.

CSPs are typically solved by a systematic backtracking search algorithm, whereas at each choice point consistency of a certain kind is maintained for constraint propagation. Many SAT and answer set solvers are based on the DP procedure (the Davis-Putnam-Logemann-Loveland algorithm) [4], where a main mechanism for space pruning is lookahead [8] - before a guess on a choice point is made, for each atom, if fixing the atom's truth value leads to a contradiction, the atom gets the opposite truth value. In this way, an atom may be propagated from already assigned atoms without going through a search process.

ASP has been advocated as an emerging paradigm of constraint programming for solving a variety of constraint problems, including CSPs [12]. It is therefore important to understand how the search space is pruned in solving CSPs. Such studies can potentially benefit both sides - an effective method in one approach may be adopted by another to improve the search efficiency. The relationship between the two becomes more interesting recently in light of the fact that answer set solvers have been integrated with CSP solvers [7]. One would expect more to come in combining the two in the future.

C. Baral et al. (Eds.): LPNMR 2005, LNAI 3662, pp. 266–278, 2005.
© Springer-Verlag Berlin Heidelberg 2005

The research direction of this paper started in [16], where it is shown that lookahead is strictly stronger than arc consistency under Niemelä's encoding with an understanding of where the added pruning power comes from.

In this paper, we extend our investigation in two fronts, one of which is to consider different encodings, and the other is to study how higher level local consistencies for CSPs such as i-consistency may be captured in lookahead. For the first goal we consider two familiar encodings in the SAT literature, the *direct encoding* and *support encoding*. In the direct encoding, disallowed tuples are expressed. We show that the pruning power of lookahead under the direct encoding is the same as the pruning power under Niemelä's encoding where allowed tuples are expressed. The pruning power of lookahead under these two encodings is precisely that of arc consistency enhanced by propagation of unique domain values to variables [16]. However, we will see that these two complementary encodings have different inference powers if tuples of literals are tested in lookahead. For the support encoding, we show that lookahead coincides with a stronger local consistency called *singleton arc consistency* in the literature [2,5]. This shows that an idea similar to lookahead had been formulated independently for CSP. We also show that by testing tuples instead of individual literals, lookahead can capture higher level consistencies, such as i-consistency and *singleton restricted path consistency* [6]. The possibility of testing n-tuples in lookahead was briefly discussed in Simons' thesis [14].

The work on relating CSP with SAT are relevant here. In [15] several encodings of CSP in SAT are compared. In general, unit propagation is weaker than arc consistency. Kasif [11] on the other hand shows that the support encoding of binary CSPs in SAT can have arc consistency established by unit propagation. Gent in [10] reports experiments that show the support encoding handles hard instances of random binary CSP more effectively than the direct encoding. Kasif's work is further extended to some higher levels and wider range of local consistencies [3]. However, none of these works consider the more powerful mechanism of lookahead in SAT solvers.

The next section defines terminologies for CSPs. Section 3 introduces answer set programming and the lookahead algorithm as defined in Smodels. Section 4 deals with direct encoding while Section 5 treats support encoding. Then, in Section 6 we discuss some relationships between the direct encoding and Niemelä's encoding [12] from CSPs to answer set programs. In Section 7 we show some experimental results that confirm some of our theoretical findings. Section 8 provides a summary.

2 Constraint Satisfaction Problem

A constraint satisfaction problem (CSP) is a triple $\mathcal{A}(X, D, C)$ where $X = \{x_1, \ldots, x_n\}$ is a finite set of variables with respective domains $D = \{D_{x_1}, \ldots, D_{x_n}\}$ listing the possible values for each variable, and C is a finite set of constraints. A constraint $c_{y_1,\ldots,y_k} \in C$ is a subset of the Cartesian product $D_{y_1} \times \ldots \times D_{y_k}$. We denote by $\mathcal{A}|_{D_x=\{a\}}$ the CSP obtained by restricting the domain D_x to $\{a\}$ in $\mathcal{A}$.

Given a CSP $\mathcal{A}(X, D, C)$, if a tuple (a, b) is in a binary constraint c_{xy}, we say a is a *support* of b w.r.t. c_{xy} and b is a *support* of a w.r.t. c_{xy}. We say x is a *neighboring variable* of y, and vice versa, if there is a constraint c_{xy} or c_{yx} in C.

An *instantiation* is a variable assignment where each variable is assigned a unique value from its domain. An assignment may be partial. A partial instantiation is *consistent* with an n-ary constraint $c \in C$ iff the assignment yields a projection of a tuple in the relation of c. A partial instantiation is consistent iff it is consistent with every constraint. A constraint $c_{y_1,...,y_k} \in C$ is *satisfied* iff the assignment to the variables $y_1, ..., y_k$ forms a tuple in the relation of $c_{y_1,...,y_k}$. A *solution* to a CSP is an instantiation of all variables that satisfy all the constraints. We denote by $x \rightarrow a$ that variable x is assigned value a from its domain.

Given a CSP $\mathcal{A}(X, D, C)$, we use n to denote the number of variables in X, e the number of constraints in C, and d the maximum domain size. In this paper we deal only with binary constraints.

A CSP is *i-consistent* iff given any consistent instantiation of any $i - 1$ variables, there exists an instantiation of any ith variable such that the i values taken together satisfy all of the constraints among the i variables. The most popular degrees of consistencies are *arc consistency* when $i = 2$, and *3-consistency* when $i = 3$ (which coincides with *path consistency* for binary constraints).

A local consistency LC is said to be *stronger than* another local consistency LC' if for any CSP in which LC holds so does LC'.

Maintaining (or enforcing) arc consistency on a CSP is a domain reduction process $-$ it removes inconsistent values from the domains of unassigned variables. This is compared with maintaining higher level consistencies where *nogood* tuples are identified. More recently, based on the idea of domain reduction, some additional notions of consistency for domain reduction are introduced [13,6], among which singleton arc consistency and singleton restricted path consistent are particularly interesting due to their potential in practical applications. We give their definitions below.

Let $\mathcal{A}(X, D, C)$ be a CSP.

- $\mathcal{A}$ is *singleton arc consistent* (SAC) iff $\forall x \in X$, $D_x \neq \emptyset$, and $\forall a \in D_x$, $\mathcal{A}|_{D_x=\{a\}}$ can be made arc consistent with non-empty domains.
- $\mathcal{A}$ is *restricted path consistent* (RPC) iff $\forall x \in X$, D_x has a non-empty arc consistent domain and, $\forall a \in D_x$ and $\forall y \in X$ such that a has a unique support $b \in D_y$, and $\forall z \in X$, $\exists c \in D_z$ such that $(a, c) \in c_{xz}$ and $(b, c) \in c_{yz}$.[1]
- $\mathcal{A}$ is *singleton restricted path consistent* (SRPC) iff $\forall x \in X$, $D_x \neq \emptyset$, and $\forall a \in D_x$, $\mathcal{A}|_{D_x=\{a\}}$ can be made restricted path consistent.

Intuitively, in order to enforce a certain kind of singleton consistency, one restricts each variable to each of its domain values and then enforces that consistency. If enforcing that consistency causes an empty domain, we then know the corresponding value of that variable cannot contribute to any solution and thus should be removed from its domain. One can see that the notion of singleton consistencies bears an idea similar to that of lookahead. In the latter, since we are dealing with Boolean variables, removing a value of a variable simply means to assign the variable with the opposite (truth) value.

Among singleton consistencies, singleton arc consistency is the most extensively studied in the literature of CSP, due to its relatively low complexity.

[1] When a constraint is not explicitly presented in C, it means all the tuples in the corresponding Cartesian product are allowed.

As *singleton* can be prefixed to any consistency, the next computationally realistic consistency to be considered is path consistency. From the definition of path consistency, two variables are required to be instantiated with all possible domain values to see if a conflict exists. To reduce the complexity, restricted path consistency is proposed where only the domain values that satisfy the stated condition in the definition are tested. In the literature, more complex singleton consistencies are considered to be only of theoretical interest [6].

3 Constraint Propagation in Smodels

Smodels implements the stable model semantics for normal logic programs which consist of rules of the form $A \leftarrow B_1, ..., B_m, \texttt{not}\ C_1, ..., \texttt{not}\ C_n.$, where A, B_i and C_i are function-free atoms, and $\texttt{not}\ C_i$ are *default negations* or simply called *not-atoms*. The head A may be the special atom $\perp$, in which case it serves as a constraint. In systems like Smodels, these programs are first instantiated to ground instances for the computation of answer sets.

An *answer set* (also called a *stable model*) is defined over the ground instantiation of a given program [9]. A set of atoms M is an answer set for a program P iff M is the least model of P^M, where P^M is defined as

$$P^M = \{a \leftarrow b_1, ..., b_m \mid a \leftarrow b_1, ..., b_m, \texttt{not}\ c_1, ..., \texttt{not}\ c_n \in P \text{ and } \forall i \in [1..n], c_i \notin M\}$$

Additional notations: Atoms and not-atoms are both called *literals*. A set of literals is consistent if there is no atom ϕ such that ϕ and $\texttt{not}\ \phi$ are both in the set.

$Atoms(\Phi)$ denotes the set of distinct atoms appearing in Φ (excluding the special atom $\perp$). The expression $not(\texttt{not}\ \phi)$ is identified with ϕ, and $not(\phi)$ is $\texttt{not}\ \phi$. Given a set of literals B, $B^+ = \{\xi \mid \xi \text{ is an atom in } B\}$ and $B^- = \{\xi \mid \texttt{not}\ \xi \in B\}$. Suppose Q is a set of atoms. Then we define $not(Q) = \{\texttt{not}\ \phi \mid \phi \in Q\}$.

Constraint propagation in Smodels is carried out by lookahead (cf. Figure 1 where P is a program and A a set of literals representing a partial truth value assignment). For each unassigned atom ϕ, lookahead assumes a truth value for it (via the function lookahead_once), if that leads to a conflict, then the opposite truth value for ϕ is in any answer set M agreeing with A (meaning $A^+ \subseteq M$ and $A^- \cap M = \emptyset$). Truth values are propagated in lookahead by a function called $expand(P, A)$ (cf. Figure 2), which returns a superset of A, representing the process of propagating the values of the atoms in A to some additional atoms. In lookahead_once, the function $conflict(P, A)$ returns true if $A^+ \cap A^- \neq \emptyset$ and false otherwise.

$Atleast(P, A)$ in the expand function returns a superset of A by repeatedly applying four propagation rules until no new literals can be deduced. Let r be a rule in program P of the form: $r = h \leftarrow a_1, ..., a_n, \texttt{not}\ b_1, ..., \texttt{not}\ b_m$. Define

$$\begin{aligned} min_r(A) &= \{h \mid \{a_1, ..., a_n\} \subseteq A^+, \{b_1, ..., b_m\} \subseteq A^-\} \\ max_r(A) &= \{h \mid \{a_1, ..., a_n\} \bigcap A^- = \emptyset, \\ &\qquad \{b_1, ..., b_m\} \bigcap A^+ = \emptyset\} \end{aligned}$$

```
Function lookahead(P, A)
  repeat
  A' := A
  A := lookahead_once(P, A)
  until A = A'
return A.
```

```
Function lookahead_once(P, A)
  B := Atoms(P) − Atoms(A)
  B := B ⋃ not (B)
  while B ≠ ∅ do
      take any literal χ ∈ B
      A' := expand(P, A ⋃ {χ})
      B := B − A'
      if conflict(P, A') then
          return expand(P, A ⋃ {not (χ)})
  end while
return A.
```

Fig. 1. Function $lookahead(P, A)$

```
Function expand(P, A)
  repeat
  A' := A
  A := Atleast(P, A)
  A := A ⋃ {not φ | φ ∈ Atoms(P) and φ ∉ Atmost(P, A)}
  until A = A'
return A.
```

Fig. 2. Function $expand(P, A)$

The four propagation rules in $Atleast(P, A)$ are:

1. If $r \in P$, then $A := A \bigcup min_r(A)$.
2. If there is an atom a such that for all $r \in P$, $a \notin max_r(A)$, then $A := A \bigcup \{not\ a\}$.
3. If an atom $a \in A$, there is only one $r \in P$ for which $a \in max_r(A)$, and there is a literal x such that $a \notin max_r(A \bigcup \{x\})$, then $A := A \bigcup \{not(x)\}$.
4. If $not\ a \in A$ and there is a literal x s.t. for some $r \in P$, $a \in min_r(A \bigcup \{x\})$, then $A := A \bigcup \{not(x)\}$.

Rule 1 adds the head of a rule to A if the body is true in A. If there is no rule with a as the head whose body is not false w.r.t. A, then a cannot be in any answer set agreeing with A. This is Rule 2. Rule 3 says that if $a \in A$, the only rule with a as the head whose body is not yet false must have its body true in extending A. Rule 4 forces the body of a rule to be false if the head is false in A.

The function $Atmost(P, A)$ is of little relevance here, since the answer set programs that encode CSPs in this paper do not have "positive loops". For these programs, it can be shown that the expand function behaves as if the fifth line in the definition were removed.

4 Constraint Propagation Under Direct Encoding

The direct encoding of a CSP by an answer set program consists of two parts. The first part specifies the *uniqueness property* - a variable in CSP is assigned exactly one value

from its domain. For each variable $x \in X$ and its domain $D_x = \{a_1, ..., a_d\}$, we use an atom $x(a_j)$ to represent that x gets the value $a_j \in D_x$. Then, for each $j \in [1..d]$ we have a rule

$$x(a_j) \leftarrow \texttt{not}\ x(a_1), ..., \texttt{not}\ x(a_{j-1}), \texttt{not}\ x(a_{j+1}), ..., \texttt{not}\ x(a_d). \quad (1)$$

These rules will be referred to as the *uniqueness encoding*.

In the second part, disallowed tuples in constraints are expressed. For each constraint $c_{xy} \in C$, and for each tuple $(a, b) \notin c_{xy}$, where $a \in D_x$ and $b \in D_y$, we have a *rule of denial*

$$\bot \leftarrow x(a), y(b). \quad (2)$$

The direct encoding of a CSP $\mathcal{A}$ is denoted $P_{dir}(\mathcal{A})$.

The correctness of the direct encoding is obvious. For the complexity, for any CSP $\mathcal{A}$, it can be verified that $O(nd^2 + ed^2)$ bounds the size of $P_{dir}(\mathcal{A})$.

4.1 Propagation Arc Consistency

Given a CSP $\mathcal{A}(X, D, C)$ and a collection Π of pairs $x \rightarrow a$, *unique value propagation* is an operation that generates an extension of Π, which is

$$\Pi \cup \{x \rightarrow a \mid c_{xy} \in C \text{ or } c_{yx} \in C, \text{ and } y \rightarrow b \in \Pi \text{ such that } a \text{ is the only value in } D_x \text{ consistent with } b\}$$

A collection of pairs Π *is closed under (unique value) propagation* if it cannot be extended further by unique value propagation. Π is said to be in *conflict* (or *conflicting*) if there are distinct values a and a' such that for some variable x, $x \rightarrow a, x \rightarrow a' \in \Pi$, otherwise it is non-conflicting.

Definition 1. *A CSP $\mathcal{A}(X, D, C)$ is* propagation arc consistent *(PAC) iff it is arc consistent, and $\forall x \in X$ and $\forall a \in D_x$, the closure of $\{x \rightarrow a\}$ under unique value propagation is non-conflicting.*

It is clear from the definition that PAC is stronger than arc consistency (AC).

Example 1. Consider a CSP with three variables x, y, and z, all with the domain $\{0, 1\}$, and the following constraints:

$$c_{xy} = \{(0, 0), (1, 1)\} \qquad c_{yz} = \{(0, 1), (1, 0)\} \qquad c_{zx} = \{(0, 0), (1, 1)\}$$

It is clear that this CSP is arc consistent. However, it is not PAC since $\{x \rightarrow 0\}$ leads to a conflict under unique value propagation.

In the direct encoding of this CSP, besides the uniqueness encoding, the disallowed tuples are expressed by the following rules:

$$\bot \leftarrow x(0), y(1). \qquad \bot \leftarrow y(0), z(0). \qquad \bot \leftarrow z(0), x(1).$$
$$\bot \leftarrow x(1), y(0). \qquad \bot \leftarrow y(1), z(1). \qquad \bot \leftarrow z(1), x(0).$$

Now, we have $\texttt{not}\ x(0) \in lookahead(P_{dir}(\mathcal{A}), \{\texttt{not}\ \bot\})$, corresponding to 0 being removed from its domain in CSP. This is obtained by assuming $x(0)$, and then by applying the expand function to infer $\texttt{not}\ y(1)$, $y(0)$, $\texttt{not}\ z(0)$, $z(1)$, and $\texttt{not}\ x(0)$, resulting in a conflict.

Under the direct encoding, the pruning power of lookahead is precisely that of PAC.

Theorem 1. *Let $\mathcal{A}(X, D, C)$ be a CSP. A value a is removed from its domain D_x by maintaining PAC on $\mathcal{A}$ iff* $\texttt{not}\ x(a) \in lookahead(P_{dir}(\mathcal{A}), \{\texttt{not}\ \bot\})$.

A proof of this theorem is relatively routine but can be lengthy. Roughly, we note that it is known that the result of lookahead is independent of the order in which literals are picked up in lookahead_once [16]. Thus, we only need to show that for any values $a_1, ..., a_k$ that are removed from $D_{x_1}, ..., D_{x_k}$, perspectively, in enforcing PAC, $lookahead(P_{dir}(\mathcal{A}), \{\texttt{not}\ \bot\})$ infers $\texttt{not}\ x_1(a_1), ..., \texttt{not}\ x_k(a_k)$, in the same order (and vice versa). Since the effect of value removals is incremental, we can apply a simple induction on k to show that if in lookahead_once $x_i(a_i)$ is picked up, the expand function generates a conflict, assuming all $\texttt{not}\ x_1(a_1), ..., \texttt{not}\ x_k(a_{i-1})$ have already been added by lookahead.

4.2 Extension to i-Consistency

Though strictly stronger than AC, it is not a surprise that PAC in general is not as powerful as higher level consistencies.

Example 2. Consider a CSP with three variables x, y, and z, all with the domain $\{0, 1, 2, 3\}$, and three constraints

$$\begin{aligned} c_{xy} &= \{(0,0),(0,1),(1,2),(1,3),(2,0),(2,1),(3,2),(3,3)\} \\ c_{xz} &= \{(0,0),(0,1),(1,2),(1,3),(2,0),(2,1),(3,2),(3,3)\} \\ c_{zy} &= \{(0,2),(1,3),(2,0),(3,1),(1,2),(0,3),(3,0),(2,1)\} \end{aligned}$$

This CSP is not path consistent: for any pair $(a_i, a_j) \in c_{xy}$ there exists no value $a_k \in D_z$ such that $(a_i, a_k) \in c_{xz}$ and $(a_k, a_j) \in c_{zy}$. However, this CSP is arc consistent. One can also check that for this CSP there is no possibility of unique value propagation. Therefore, it is PAC.

Lookahead may be extended to test tuples of literals instead of single literals [14]. Here we show that i-consistency can be captured by testing $(i-1)$-tuples.

Theorem 2. *Let $\mathcal{A}(X, D, C)$ be a CSP. For any $1 < i \leq |X|$, let $\{y_1, ..., y_{i-1}\}$ be any $i-1$ variables in X and y_i be any ith variable. For any consistent instantiation $\Pi = \{y_1 \to b_1, ..., y_{i-1} \to b_{i-1}\}$, if y_i has no instantiation consistent with Π, then a conflict will be generated in* $expand(P_{dir}(\mathcal{A}), \{\texttt{not}\ \bot, y_1(b_1), ..., y_{i-1}(b_{i-1})\})$.

Note that although this theorem is not stated directly using lookahead, it is clear that what it states is that any inconsistency in enforcing i-consistency can be detected by testing the corresponding $(i-1)$-tuple by lookahead. This implies that if lookahead tests all $(i-1)$-tuples, and the result of lookahead in such testing is independent of any order in which these tuples are tested, then i-consistency is enforced.

Proof. That there is no instantiation of y_i that is consistent with Π implies that for any $v \in D_{y_i}$, there exists $y_j \in \{y_1, ..., y_{i-1}\}$, such that $b_j \in D_{y_j}$ is inconsistent with v.

Thus, in $P_{dir}(\mathcal{A})$ there must be a rule of denial $\bot \leftarrow y_j(b_j), y_i(v)$. Since $y_j(b_j)$ is in the input set, $\texttt{not}\ y_i(v)$ can be inferred by Rule 4 of the Atleast function. Since this is the case for any $v \in D_{y_i}$, a conflict will be generated by the uniqueness encoding in $P_{dir}(\mathcal{A})$. □

5 Constraint Propagation Under Support Encoding

The support encoding of a CSP by an answer set program consists of two parts. The first part is the uniqueness encoding, the same as in the direct encoding. The difference lies in the second part: for each constraint $c_{xy} \in C$ and for each value $a \in D_x$, let $\{b_1, ...b_k\}$ be the set of the supports for a w.r.t. c_{xy}, the *support rule* for a w.r.t. c_{xy} is

$$\bot \leftarrow x(a), \texttt{not}\ y(b_1), ..., \texttt{not}\ y(b_k). \tag{3}$$

Similarly, for each value b in D_y, let $\{a_1, ...a_{k'}\}$ be the set of the supports for b w.r.t. c_{xy}, the *support rule* for b w.r.t. c_{xy} is

$$\bot \leftarrow y(b), \texttt{not}\ x(a_1), ..., \texttt{not}\ x(a_{k'}). \tag{4}$$

The support encoding of a CSP $\mathcal{A}$ is denoted $P_{sup}(\mathcal{A})$.

Example 3. Let a CSP have two variables x and y with domains $D_x = D_y = \{0, 1\}$, and a constraint $c_{xy} = \{(0, 0), (0, 1), (1, 0)\}$. The support rules for c_{xy} are:

$$\bot \leftarrow x(0), \texttt{not}\ y(0), \texttt{not}\ y(1). \qquad \bot \leftarrow y(0), \texttt{not}\ x(0), \texttt{not}\ x(1).$$
$$\bot \leftarrow x(1), \texttt{not}\ y(0). \qquad \bot \leftarrow y(1), \texttt{not}\ x(0).$$

It can be seen that the size of the resulting program by support encoding remains to be bounded by $O(nd^2+ed^2)$. Again, the correctness of the support encoding is obvious.

Our goal in this section is to show that, under the support encoding, lookahead coincides with singleton arc consistency (SAC). First, we should place this result in a context, namely the fact that SAC is strictly stronger than propagation arc consistency (PAC) introduced in the last section. It is easy to see that whenever unique value propagation is possible, enforcing AC will carry it out. Thus, SAC is stronger than PAC. Example 4 below shows a CSP which is PAC but not SAC.

Example 4. Consider a CSP with three variables all with domain $D_x = D_y = D_z = \{0, 1, 2, 3\}$, and three constraints

$$c_{xy} = \{(0, 0), (0, 1), (1, 0), (1, 1), (2, 2), (2, 3), (3, 2), (3, 3)\}$$
$$c_{yz} = \{(0, 0), (0, 1), (1, 0), (1, 1), (2, 2), (2, 3), (3, 2), (3, 3)\}$$
$$c_{zx} = \{(0, 2), (0, 3), (1, 2), (1, 3), (2, 0), (2, 1), (3, 0), (3, 1)\}$$

This CSP is arc consistent, and there is no unique value propagation; hence it is PAC. But it is not SAC. Suppose we restrict D_x to $\{0\}$. Enforcing AC removes 2 and 3 from D_y as well as from D_z, resulting in 0 being removed from D_x.

We therefore have

Theorem 3. *SAC is strictly stronger than PAC.*

Now, let us use the following example to explain the main technical result of this section.

Example 5. Consider a CSP $\mathcal{A}$ with three variables all with domain $\{0, 1\}$, and three constraints:

$$c_{xy} = \{(0,0),(0,1),(1,1)\} \quad c_{yz} = \{(0,0),(1,1)\} \quad c_{zx} = \{(0,1),(1,0)\}$$

Enforcing arc consistency on $\mathcal{A}|_{D_x=\{1\}}$ will eventually remove 1 from D_x. On the other hand, we have $\texttt{not}\ x(1) \in \mathit{lookahead}(P_{sup}(\mathcal{A}), \{\texttt{not}\ \bot\})$.

The following lemma is essential in proving the main result.

Lemma 1. *Let $\mathcal{A}(X, D, C)$ be a CSP. For any value $a \in D_x$, a is removed by enforcing arc consistency iff $\texttt{not}\ x(a) \in \mathit{Atleast}(P_{sup}(\mathcal{A}), \{\texttt{not}\ \bot\})$.*

The lemma can be proved by an induction on the sequence of value removals. For the base case, for any constraint c_{xy}, if a value $a \in D_x$ does not have a support in D_y, the corresponding support rule in $P_{sup}(\mathcal{A})$ is $\bot \leftarrow x(a)$. By Rule 4 of $\mathit{Atleast}$ we infer $\texttt{not}\ x(a)$. For the inductive step, suppose a value a in D_x is removed because all of its supports in D_y have already been removed. Then, the corresponding rule in $P_{sup}(\mathcal{A})$ is

$$\bot \leftarrow x(a), \texttt{not}\ y(b_1), ..., \texttt{not}\ y(b_k).$$

Again by Rule 4 of the Atleast function, because all $\texttt{not}\ y(b_i)$, $i \in [1..k]$ are already inferred, we must have $\texttt{not}\ x(a)$. The argument for the other direction is similar.

This lemma leads to the following theorem.

Theorem 4. *Let $\mathcal{A}(X, D, C)$ be a CSP. A value a is removed from its domain D_x by maintaining SAC on $\mathcal{A}$ iff $\texttt{not}\ x(a) \in \mathit{lookahead}(P_{sup}(\mathcal{A}), \{\texttt{not}\ \bot\})$.*

We can further show that by testing pairs of literals, lookahead captures singleton restricted path consistency (SRPC).

Theorem 5. *Let $\mathcal{A}(X, D, C)$ be a CSP. For any constraint $c_{xy} \in C$, if a pair $(a, b) \in c_{xy}$ is identified as* nogood *by maintaining RPC on $\mathcal{A}|_{D_z=\{v\}}$, then either $\mathit{expand}(P_{sup}(\mathcal{A}), \{\texttt{not}\ \bot, z(v), x(a)\})$ or $\mathit{expand}(P_{sup}(\mathcal{A}), \{\texttt{not}\ \bot, z(v), y(b)\})$ generates a conflict.*

Note that, in enforcing SRPC, inconsistencies are detected by fixing three values, one for domain restriction and the other two for testing a path. Since one of the latter two is a unique support of the other, it can be propagated in the Atleast function. This is why lookahead only needs to test two values.

6 Relationships

Niemelä proposes to encode a CSP by representing allowed tuples [12]. The resulting program consists of the uniqueness encoding and the rules for allowed tuples: for each constraint c_{xy}, and each pair $(a, b) \in c_{xy}$, we have

$$\mathit{sat}(c_{xy}) \leftarrow x(a), y(b). \tag{5}$$

where $sat(c_{xy})$ is a new atom. We then ask an answer set generator to compute the answer sets that contain all $sat(c_{xy})$ where $c_{xy} \in C$.

It is shown in [16] that under Niemelä's encoding lookahead coincides with PAC. However, the following example shows that Theorem 2 does not hold under Niemelä's encoding, illustrating that these two complementary encodings behave differently when testing tuples of literals.

Example 6. Given three variables and their domains $D_x = D_y = \{0\}$ and $D_z = \{0, 1, 2, 3\}$, and three constraints

$$c_{xy} = \{(0,0)\} \;\; c_{xz} = \{(0,0),(0,1)\} \;\; c_{yz} = \{(0,2),(0,3)\}$$

$\{x \to 0, y \to 0\}$ cannot be consistently extended to z. Under the direct encoding, we have rules of denial as follows

$$\bot \leftarrow x(0), z(2). \quad \bot \leftarrow x(0), z(3). \quad \bot \leftarrow y(0), z(0). \quad \bot \leftarrow y(0), z(1).$$

When we test the pair $\langle x(0), y(0)\rangle$, we derive `not` $z(0)$, `not` $z(1)$, `not` $z(2)$, and `not` $z(3)$, from which conflicts are derived by the uniqueness encoding.

However, one can check that no conflict can be derived by the expand function under Niemelä's encoding. The rules for allowed tuples in Niemelä's encoding are:

$$\begin{array}{ll} sat(c_{xy}) \leftarrow x(0), y(0). & \\ sat(c_{xz}) \leftarrow x(0), z(0). & sat(c_{xz}) \leftarrow x(0), z(1). \\ sat(c_{yz}) \leftarrow y(0), z(2). & sat(c_{yz}) \leftarrow y(0), z(3). \end{array}$$

When we test the pair $\langle x(0), y(0)\rangle$, for $sat(c_{xz})$ to be true one of the literals in $\{z(0), z(1)\}$ must be true; for $sat(c_{yz})$ to be true one of the literals in $\{z(2), z(3)\}$ must be true. But their intersection is empty. To arrive at the conclusion of conflict, one has to reason disjunctively along with the uniqueness encoding. But the expand function of Smodels does not perform this type of reasoning.

7 Experiments

The experiments were designed to test the following: the difference in performance of the direct and support encodings from CSP to ASP.

We ran the direct encoding and support encoding on a wide range of problems varying the number of variables, the size of the domains, the number of constraints and the number of no-goods. We found that problems with approximately 10 variables are best for testing, as increasing the number of variables can significantly extend the running time of the program for the more complex problems. We then tested with various domain sizes and made the number of constraints range from 5 to 45. The number of no-good values as a percentage of the constraints is called the tightness, which in our experiments ranged from 5% to 95%.

When measuring the running time of the ASP programs, we only report the time required to run Smodels and omit the time required to ground the problem through *lparse*. We do this because it is possible to directly translate from the CSP instance to a

grounded instance without having to use a grounding program. All times were recorded using the *time* command in Linux, and recording the user CPU time.

All experiments were performed by generating random CSP instances using the generator found at [1]. All results reported were performed on an Intel Pentium 4 2.00 Ghz machine with 512 Mb of RAM running Slackware Linux 10.0.

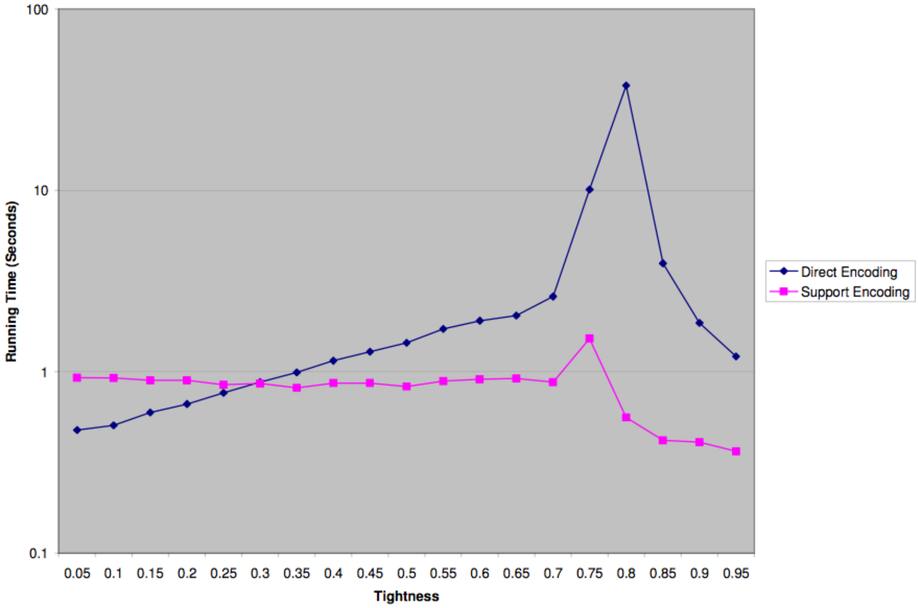


Fig. 3. Run time vs. tightness

Figure 3 shows the running time (in seconds) of Smodels plotted against the tightness of the problems. The tests were performed on problems with 10 variables, a domain size of 30, and 20 constraints. Each point on the graph represents the average running time of 10 random instances. The direct encoding begins by having a better running time than the support encoding, but is quickly outdone by the support encoding as the tightness of the problems increase. This is especially apparent as the problem set is about to enter its phase transition; the support encoding is greatly outperforming direct encoding.

It is worth noting that Gent performed similar tests between the direct and support encoding of CSP to SAT, using Chaff which does not employ the lookahead mechanism. In his tests a similar shape to the graph was reported. This indicates that the support encoding has a similar improvement in running time over the direct encoding in both the SAT and ASP encodings, with or without lookahead.

We also decided to measure the amount of searching that both the direct and support encodings must do in order to arrive at a solution. This is measured in Smodels via the number of picked atoms reported. This number represents the number of atoms that the

lookahead function must pick a value for and then test to see if any conflicts occur. It is a good (though implicit) measure of the amount of search that occurs.

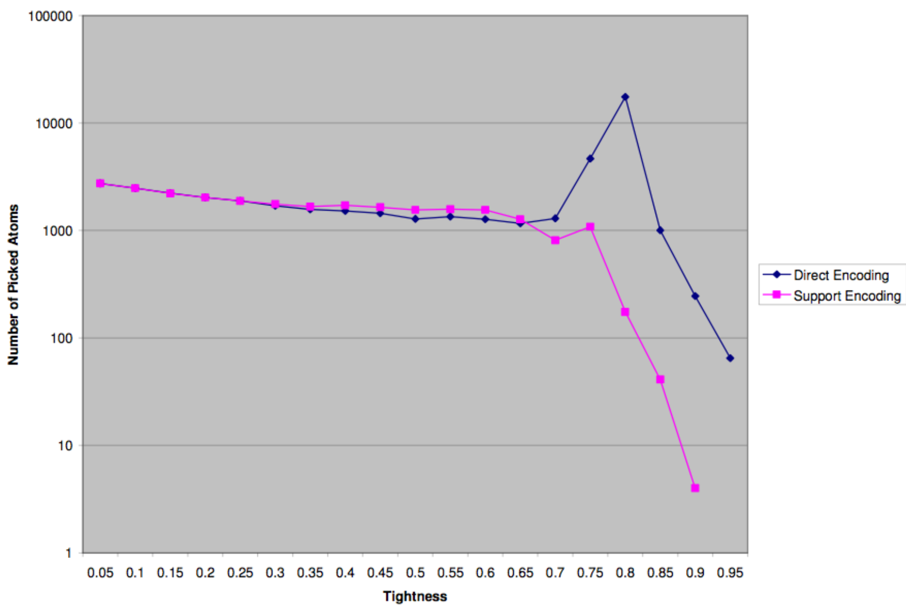


Fig. 4. Picked atoms vs. tightness

Figure 4 shows the number of picked atoms plotted against the tightness of the problem set. The problem set is identical to the one used for figure 3. For lower values of tightness, the number of picked atoms is almost identical in both the direct and support encoding. Similar to Figure 3, as the problem set begins to enter its phase transition, the number of picked atoms between the two encodings begins to diverge. The support encoding clearly picks less atoms as the problem set becomes tighter.

8 Conclusion

We summarize below the main findings reported in this paper.

(1) Lookahead has the same pruning power under the direct encoding as well as under Niemelä's encoding (Theorem 1 and [16]).

(2) When testing $(i-1)$-tuples, lookahead captures i-consistency under the direct encoding (Theorem 2).

(3) Under the support encoding, lookahead coincides with SAC (Theorem 4).

(4) Lookahead performs more effectively under the support encoding than under the direct encoding (Theorems 1, 4 and 3). Experimentally, the former shows a substantial improvement in running time than the latter on hard random binary CSPs.

(5) When testing pairs of literals, lookahead captures singleton restricted path consistency under the support encoding (Theorem 5).

(6) Lookahead provides an uniform algorithm for maintaining local consistencies of various kinds, in particular enforcing both SAC and PAC is of the complexity $O(n^3d^4 + en^2d^4)$. This is because the number of atoms in these encodings is nd, lookahead may be called at most $O(n^2d^2)$ times, and each time expand takes time linear in the size of the program (which is bounded by $O(nd^2 + ed^2)$). It is interesting to see that this complexity is comparable to that of the algorithm specifically designed for SAC in [5].

Acknowledgment: We thank the referees for their comments, which have been useful in improving the presentation of the paper.

References

1. C. Bessière. Uniform random binary csp generator. Retrieved December 19, 2004, Website www.lirmm.fr/∼bessiere/generator.html.
2. C. Bessière and R. Debruyne. Theoretical analysis of singleton arc consistency. In *Proc. ECAI'04 Workshop on Modeling and Solving Problems with Constraints*, pages 20–29, 2004.
3. C. Bessière, E. Hebrard, and T. Walsh. Local consistencies in SAT. In *Proc. Int'l Conf. on Theory and Applications of Satisfiability Testing*, pages 400–407, 2003.
4. M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Communications of the ACM*, 5(7):394–397, July 1962.
5. R. Debruyne and C. Bessière. Some practical filtering techniques. In *Proc. IJCAI'97*, pages 412–417, 1997.
6. R. Debruyne and C. Bessière. Domain filtering consistencies. *J. Artificial Intelligence Research*, 14:205–230, 2001.
7. Islam Elkabani, Enrico Pontelli, and Tran Cao Son. Smodels with CLP and its applications: a simple and effective approach to aggregates in ASP. In *Proc. ICLP'04*, pages 73–89, 2004.
8. J.W. Freeman. *Improvements to propositional satisfiability search algorithms*. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, 1995.
9. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proc. 5th ICLP*, pages 1070–1080. MIT Press, 1988.
10. Ian Gent. Arc consistency in SAT. In *Proc. ECAI 2003*, pages 121–125, 2002.
11. S. Kasif. On the parallel complexity of discrete relaxation in constraint satisfaction networks. *Artificial Intelligence*, pages 275–286, 1990.
12. I. Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Math. and Artificial Intelligence*, 25(3-4):241–273, 1999.
13. P. Prosser, K. Stergiou, and T. Walsh. Singleton consistencies. In *Proc. CP'00*, pages 353–368, 2000.
14. P. Simons. *Extending and Implementing the Stable Model Semantics*. PhD thesis, Helsinki University of Technology, Helsinki, Finland, 2000.
15. T. Walsh. CSP vs. SAT. In *Proc. Principles and Practice of Constraint Programming*, pages 441–456, 2000.
16. J. You and G. Hou. Arc consistency + unit propagation = lookahead. In *Proc. ICLP'04*, pages 314–328, 2004.

Nested Epistemic Logic Programs

Kewen Wang[1] and Yan Zhang[2]

[1] Griffith University, Australia
k.wang@griffith.edu.au
[2] University of Western Sydney, Australia
yan@cit.uws.edu.au

Abstract. Nested logic programs and epistemic logic programs are two important extensions of answer set programming. However, the relationship between these two formalisms is rarely explored. In this paper we first introduce the epistemic HT-logic, and then propose a more general extension of logic programs called *nested epistemic logic programs*. The semantics of this extension - named *equilibrium views* - is defined on the basis of the epistemic HT-logic. We prove that equilibrium view semantics extends both the answer sets of nested logic programs and the world views of epistemic logic programs. Therefore, our work establishes a unifying framework for both nested logic programs and epistemic logic programs. Furthermore, we also provide a characterization of the strong equivalence of two nested epistemic logic programs.

1 Introduction

Answer set programming (ASP) [6] was developed in the late of 1990s and has been widely recognized as a promising tool for effective knowledge representation and declarative problem solving [1]. ASP is based on the *answer set semantics* of logic programs introduced by Gelfond and Lifschitz [4,5]. The formal systems for ASP may have different features such as default negation, explicit negation, disjunction and preference. Normal, general, extended and disjunctive logic programs are among the major ASP formalisms.

As many researchers (including [3,8]) have noticed, the languages of logic programming are still insufficient in representing commonsense knowledge. Recently answer set semantics has been extended to nested logic programs [8], in which arbitrarily nested formulas are allowed. On the other hand, ASP is also expanded to epistemic logic programs by Gelfond [3], where belief operators can be explicitly presented so that incomplete information may be correctly represented in the extent of multiple belief sets. The semantics of epistemic logic programs is defined as the collection of its *world views*, which are generalizations of the answer sets for logic programs without nested expressions.

Having examined the syntax and semantics of nested logic programs and epistemic logic programs, people may observe an important fact: Although the world view semantics of epistemic logic programs generalizes the answer set semantics for disjunctive (extended) logic programs, it, however, cannot be used as the semantics for the epistemic logic programs with nested expressions containing belief operators. Hence, the following two problems remain open:

C. Baral et al. (Eds.): LPNMR 2005, LNAI 3662, pp. 279–290, 2005.
© Springer-Verlag Berlin Heidelberg 2005

1. As two extensions of ASP, can nested logic programs and epistemic logic programs be unified in one common language?
2. Can nested expressions of formulas with belief operators be allowed in both the head and body of rules in an epistemic logic program?

We observe that existing nonmonotonic epistemic logics, such as [9], do not provide direct solutions to the above problems. As it will be illustrated in the following, since the world view semantics is defined based on a transformation from epistemic logic programs to disjunctive logic programs by eliminating the belief operators in the body of rules, it is not feasible to simply allow belief operators to occur in the head of rules in an epistemic logic program. Hence, it seems to be inevitable to develop a new approach to solve these problems instead of seeking for a straightforward extension of Gelfond's world view semantics.

This paper aims to solve these problems in a unified manner. In particular, we first introduce a new logic called *epistemic equilibrium HT-logic*. This logic is a natural integration of the equilibrium HT-logic [10,11] and modal logic. Based on this logic, we then specify a more general extension of logic programs called *nested epistemic logic programs (NELPs)*. The semantics of this extension - named *equilibrium views* - is defined on the basis of the epistemic HT-logic. We prove that equilibrium view semantics extends both the answer sets of nested logic programs and the world views of epistemic logic programs. Therefore, our work establishes a unifying framework for both nested logic programs and epistemic logic programs. Furthermore, we also provide a characterization of the strong equivalence of two nested epistemic logic programs.The main results of this paper are summarized as follows:

1. Equilibrium view semantics extends the answer set semantics of nested logic programs;
2. Equilibrium view semantics extends the world view semantics of epistemic logic programs;
3. Two nested epistemic logic programs are strong equivalent if and only if they are equivalent in the epistemic HT-logic.

The rest of this paper is organized as follows. Section 2 proposes a new logic called epistemic HT logic and defines its semantics. Section 3 presents a new logic programming language (i.e. nested epistemic logic programs) which extends both the nested logic programs and epistemic logic programs. Section 4 proves two major results to show that the equilibrium view semantics generalizes both the answer set semantics for nested logic programs and the world view semantics for epistemic logic programs. Section 5 further proves a result about the strong equivalence of nested epistemic logic programs. Section 6 discusses how to add the second negation into the new class of programs. Finally, section 7 concludes the paper with some discussions.

2 Epistemic HT-Logic and Equilibrium Models

2.1 Syntax and Semantics

The language of our epistemic HT-logic will be the modal language which extends the classical propositional language by means of two belief operators K and M. We

consider classical propositional formulas built from propositional atoms and the 0-place connective $\bot$ ("false") using the binary connectives $\vee$, $\wedge$ and $\rightarrow$. We use $\top$ for $\bot \leftarrow \bot$, and $\neg F$ for $F \rightarrow \bot$ where F is a formula. Modal formulas are obtained by addition of the following clauses to the usual inductive definition of propositional formulas:

- If F is a formula, then KF is a formula;
- If F is a formula, then MF is a formula.

KF is read as "F is known to be true" and MF is read as "F may be believed to be true". If a is an atom in the classical propositional logic, then an objective literal is either a or $\neg a$, and both Kb and Mb are called *subjective literals* for any objective literal b. Similarly, F is a *subjective formula* if F contains at least one belief operator. An *epistemic theory* is a (finite) set of formulas in the language of epistemic HT-logic.

The HT-logic (i.e. the logic of here-and-there) is also known as "the logic of present and future", which is basically a three-valued logic. Pearce first used this logic to characterize the answer set semantics of logic programs [11]. More recently, Lifschitz, Pearce and Valverde characterize the strong equivalence of logic programs through the HT-logic [7]. In the following we extend the semantics of the HT logic to the epistemic HT-logic. So in our logic, we will have two tenses (H and T) and two belief operators (K and F).

Let $\mathcal{A}$ be a collection of sets of (ground) atoms. An *epistemic HT-interpretation* is defined as an ordered tuple $(\mathcal{A}, I^H, I^T)$ where I^H, I^T are sets of atoms with $I^H \subseteq I^T$. If $I^H = I^T$, we say $(\mathcal{A}, I^H, I^T)$ is *total*. Notice that we do not require $I^H \in \mathcal{A}$ or $I^T \in \mathcal{A}$.

For any epistemic HT-interpretation $(\mathcal{A}, I^H, I^T)$, any tense $t \in \{H, T\}$, and any formula F, we define when $(\mathcal{A}, I^H, I^T, t)$ *satisfies* F, denoted as $(\mathcal{A}, I^H, I^T, t) \models F$, as follows:

- for any atom F, $(\mathcal{A}, I^H, I^T, t) \models F$ if $F \in I^t$.
- $(\mathcal{A}, I^H, I^T, t) \not\models \bot$.
- $(\mathcal{A}, I^H, I^T, t) \models KF$ if $(\mathcal{A}, J^H, J^T, t) \models F$ for all $J^H, J^T \in \mathcal{A}$ with $J^H \subseteq J^T$.
- $(\mathcal{A}, I^H, I^T, t) \models MF$ if $(\mathcal{A}, J^H, J^T, t) \models F$ for some pair $J^H, J^T \in \mathcal{A}$ with $J^H \subseteq J^T$.
- $(\mathcal{A}, I^H, I^T, t) \models F \wedge G$ if $(\mathcal{A}, I^H, I^T, t) \models F$ and $(\mathcal{A}, I^H, I^T, t) \models G$.
- $(\mathcal{A}, I^H, I^T, t) \models F \vee G$ if $(\mathcal{A}, I^H, I^T, t) \models F$ or $(\mathcal{A}, I^H, I^T, t) \models G$.
- $(\mathcal{A}, I^H, I^T, t) \models F \rightarrow G$ if, for every tense t' with $t \leq t'$, $(\mathcal{A}, I^H, I^T, t') \not\models F$ or $(\mathcal{A}, I^H, I^T, t') \models G$.
- $(\mathcal{A}, I^H, I^T, t) \models \neg F$ if $(\mathcal{A}, I^H, I^T, t) \models F \rightarrow \bot$

It is easy to see that if F does not contain any belief operators, $(\mathcal{A}, I^H, I^T, t) \models F$ is irrelevant to the collection $\mathcal{A}$; if F is a subjective literal (either Ka or Ma), $(\mathcal{A}, I^H, I^T, t) \models F$ is irrelevant to I^H and I^T. For example, take $\mathcal{A} = \{\{a, b\}, \{a, c\}\}$ and then $(\mathcal{A}, I^H, I^T, t) \models Ka$ for any sets of atoms I^H and I^T with $I^H \subseteq I^T$.

Epistemic HT-logic has the following basic properties which will be used in subsequent sections.

Lemma 1. *For any epistemic HT-interpretation* $(\mathcal{A}, I^H, I^T)$*, if* $(\mathcal{A}, I^H, I^T, H) \models F$*, then* $(\mathcal{A}, I^H, I^T, T) \models F$*.*

The intuition behind this lemma is obvious: If a statement is true "here", then it is also true "there". This property is guaranteed by the condition $I^H \subseteq I^T$.

Finally, we say that an epistemic HT-interpretation $(\mathcal{A}, I^H, I^T)$ *satisfies* a formula F, denoted $(\mathcal{A}, I^H, I^T) \models F$, if $(\mathcal{A}, I^H, I^T, H) \models F$.

A *model* of an epistemic theory E is an epistemic HT-interpretation $(\mathcal{A}, I^H, I^T)$ by which every formula in E is satisfied.

2.2 Epistemic Equilibrium Logic

Pearce's equilibrium logic is a kind of minimal model reasoning based on the HT-logic and its semantics is defined as the set of the equilibrium models [11]. An interesting result about the equilibrium logic is that it provides a characterization of the answer sets for nested logic programs [7]. In this subsection, we generalize the notion of equilibrium models to our epistemic HT-logic and the resulting logic is called *epistemic equilibrium logic*.

Definition 1. *An* epistemic equilibrium model *of an epistemic theory Γ is a total epistemic HT-interpretation $(\mathcal{A}, I, I)$ such that*

(i) *$(\mathcal{A}, I, I)$ is a model of Γ.*
(ii) *for every proper subset J of I, $(\mathcal{A}, J, I)$ is not a model of Γ.*

Epistemic equilibrium logic is the logic whose semantics is defined through epistemic equilibrium models. To provide a unifying characterization for the semantics of both nested logic programs and epistemic logic programs, we need the following definition.

Definition 2. *Let $\mathcal{A}$ be a collection of sets of atoms occurring in an epistemic theory Π. We say $\mathcal{A}$ is an equilibrium view if $\mathcal{A}$ is a maximal collection that satisfies*

$$\mathcal{A} = \{I \mid (\mathcal{A}, I, I) \textit{ is an equilibrium model of } \Pi\}.$$

To illustrate our definitions, let us look at the following examples.

Example 1. Let Π_1 be the epistemic theory containing only the formula $\{Ka \vee b\}$. Then both $(\{\{a\}\}, \{a\}, \{a\})$ and $(\{\{b\}\}, \{b\}, \{b\})$ are epistemic equilibrium models of Π_1. Moreover, $\mathcal{A}_1 = \{\{a\}\}$ and $\mathcal{A}_2 = \{\{b\}\}$ are equilibrium views of Π_1. But $\mathcal{A}_3 = \{\{a\}, \{b\}\}$ is not an equilibrium view of Π_1 since $(\mathcal{A}_3, \{a\}, \{a\})$ is not an epistemic model of Π_1 (note that $(\mathcal{A}_3, \{a\}, \{a\}) \not\models Ka$). $\mathcal{A}_4 = \{\{a, b\}\}$ is not an equilibrium view of Π_1 either because $(\mathcal{A}_4, \{a, b\}, \{a, b\})$ is not an epistemic equilibrium model.

Example 2. Let $\Pi_2 = \{K(a \vee b)\}$ be an epistemic theory. Then $\{\{a\}, \{b\}\}$ is the unique equilibrium view of Π_2. Note that although $(\{\{a\}\}, \{a\}, \{a\})$ and $(\{\{b\}\}, \{b\}, \{b\})$ are epistemic equilibrium models of Π_1, neither $\{\{a\}\}$ nor $\{\{b\}\}$ is an equilibrium view of Π_2 since they are not maximal.

Example 3. Let $\Pi_3 = \{a, Ka \rightarrow b \vee c\}$ be an epistemic theory. It is easy to see that Π_3 has a unique equilibrium view $\{\{a, b\}, \{a, c\}\}$.

It is worth to mentioning that differently from the standard propositional modal logic, for instance S5, the epistemic equilibrium logic can be viewed as a kind of minimal model reasoning about epistemic concepts (i.e knowledge and belief). In this way, the equilibrium view semantics shares the same spirit of Gelfond's world view semantics. However, as will be shown next, epistemic equilibrium logic is general enough to characterize the semantics of nested epistemic logic programs, while the world view semantics cannot.

3 Nested Epistemic Logic Programs (NELPs)

As we will see in the next section, the epistemic HT-logic is actually a very general extension of both nested logic programs (NLPs) and epistemic logic programs (ELPs). To make this comparison more direct, we generalize both the syntax of NLPs and the syntax of ELPs by introducing a class of logic programs called *nested epistemic logic programs* or *NELP*. This language corresponds to a subset of the language of the epistemic HT-logic.

The *atom* is understood as in propositional logic. *Elementary formulas* are propositional atoms and the 0-place connective $\bot$ ("false") and $\top$ ("true"). *NELP formulas* are built from elementary formulas using negation as failure "*not*", conjunction ",", disjunction ";", and the two belief operators K and M.

An *NELP rule* is an expression of the form

$$F \leftarrow G$$

where F and G are NELP formulas called the *head* and the *body* of the rule. For any rule r, its head and body are denoted $head(r)$ and $body(r)$, respectively. A *nested epistemic logic program (abbreviated NELP)* is a (finite) set of NELP rules.

For our purpose, in this paper we will only consider propositional epistemic logic programs where rules containing variables are viewed as the set of all ground rules by replacing these variables with all constants occurring in the language.

Let us think of NELP rules as epistemic formulas by replacing every "*not*" with "$\neg$", every comma with "$\wedge$", every semicolon ";" with disjunction "$\vee$", and transforming every rule $head \leftarrow body$ into the implication $body \rightarrow head$. Accordingly we can turn every nested epistemic logic program Π into an epistemic theory. When no confusion is caused, we will not distinguish a nested epistemic logic program Π and its corresponding epistemic theory. Note that the negation "$\neg$" corresponds to the negation as failure rather than the strong negation (or classical negation) in logic programming. For simplicity, the second negation will not be considered until in Section 6. For example, we may use Π to denote both the NELP $\{Ka; b \leftarrow c, Md, not e\}$ and its corresponding epistemic theory $\{c \wedge Md \wedge \neg e \rightarrow Ka \vee b\}$.

Now based on the epistemic HT-logic introduced in Section 2, we define the semantics of nested epistemic logic programs as follows.

Definition 3. *Let Π be a nested epistemic logic program and $\mathcal{A}$ be a collection of sets of atoms. We say $\mathcal{A}$ is an* equilibrium view *of Π if $\mathcal{A}$ is an equilibrium view of the corresponding epistemic theory Π.*

Example 4. Consider the nested epistemic logic program Π:

$Ka; Kb \leftarrow,$
$c \leftarrow Ka, not\, Mb,$
$d \leftarrow Kb, not\, Ma.$

It is easy to see that Π has two equilibrium views $\{\{a, c\}\}$ and $\{\{b, d\}\}$. Note that if we change the first rule in Π to be $a; b \leftarrow$, then the modified program will only have one equilibrium view $\{\{a\}, \{b\}\}$.

4 Epistemic Equilibrium Logic and Logic Programs

The class of NELPs contains two major classes of logic programs: *nested logic programs* and *epistemic logic programs*. Thus NELPs generalize most classes of logic programs including normal logic programs and disjunctive logic programs. In this section we will prove that the equilibrium view semantics of NELPs extends both the world view semantics of epistemic logic programs and answer set semantics of nested logic programs.

4.1 Equilibrium Views and Answer Sets of Nested Logic Programs

An *NLP rule* is a special NELP rule which contains no belief operators, and a *nested logic program (NLP)* is a set of NLP rules. Similarly, an *NLP* formula is an NELP formula containing no belief operators.

To define the answer sets of nested logic programs, we first define when a set S of atoms *satisfies* an NLP formula F, denoted as $S \models F$, recursively as follows:

$S \models F$ if F is an atom and $F \in S$,
$S \models \top$,
$S \models \bot$,
$S \models (F, G)$ if $S \models F$ and $S \models G$,
$S \models (F; G)$ if $S \models F$ or $S \models G$,
$S \models not\, F$ if $S \not\models F$.

We say a nested logic program Π is *closed* under a set of atoms S if, for every rule r, $body(r)$ implies $head(r)$. Then the definition of *answer sets* is defined in two steps:

- Let Π be a nested logic program without negation as failure *not*. A set S of atoms is an *answer set* of Π if S is minimal set closed under S;
- For an arbitrary nested logic program Π, the *reduct* Π^S with respect to a set S of atoms is obtained by replacing every maximal occurrence of a formula of the form $not\, F$ in Π with $\bot$ if $S \models F$ and with $\top$ if $S \models not\, F$. We say S is an *answer set* of Π if S is an answer set of Π^S.

As we have noted before, a nested logic program is also a nested epistemic logic program and thus any nested logic program Π can be assigned two semantics: *answer sets* and *equilibrium views*. The following result states that these two semantics coincide for nested logic programs and thus the equilibrium view semantics generalizes the answer set semantics.

Theorem 1. *Let Π be a nested logic program and S be a set of atoms. Then S is an answer set of Π if and only if there exists an equilibrium view $\mathcal{A}$ of Π (as a nested epistemic logic program) such that $S \in \mathcal{A}$.*

Proof. ($\Rightarrow$) If S is an answer set of Π, let $\mathcal{A}$ denote the set of minimal models of Π^S. Then we have $S \in \mathcal{A}$ by the definition of answer sets. We need only to prove that $\mathcal{A}$ is an equilibrium view of Π (as a nested epistemic logic program). Since Π contains no belief operators, $(\mathcal{A}, S, S)$ is an equilibrium model of Π in the epistemic HT-logic if and only if (S, S) is an equilibrium model of Π in ordinary HT-logic. Again, by Lemma 3 in [7], (S, S) is an equilibrium model of Π in ordinary HT-logic if and only if S is an answer set of Π. Thus $(\mathcal{A}, S, S)$ is an equilibrium model of Π in the epistemic HT-logic.
($\Leftarrow$) Using the above argument, we have that if $(\mathcal{A}, S, S)$ is an equilibrium model of Π in the epistemic HT-logic then S is an answer set of Π.

4.2 Equilibrium Views and World Views of Epistemic Logic Programs

Epistemic logic programs were first proposed by Gelfond [3] in order to overcome difficulties in reasoning about disjunctive information through disjunctive logic programs. It turns out that epistemic logic programs can be used as an effective formulation to represent and reason about agents' epistemic states and hence have great potential in agent programming. The semantics for epistemic logic programs is based on the pair $(\mathcal{A}, S)$, where $\mathcal{A}$ is a collection of sets of ground literals and S is a set in $\mathcal{A}$. The truth of an NELP formula F in $(\mathcal{A}, S)$ is denoted by $(\mathcal{A}, S) \models F$ and the falsity by $(\mathcal{A}, S) =\!| F$, and are defined as follows: $(\mathcal{A}, S) \models F$ iff $F \in S$ where F is a ground atom.

$(\mathcal{A}, S) \models KF$ iff $(\mathcal{A}, S_i) \models F$ for all $S_i \in \mathcal{A}$.
$(\mathcal{A}, S) \models MF$ iff $(\mathcal{A}, S_i) \models F$ for some $S_i \in \mathcal{A}$.
$(\mathcal{A}, S) \models (F, G)$ iff $(\mathcal{A}, S) \models F$ and $(\mathcal{A}, S) \models G$.
$(\mathcal{A}, S) \models (F; G)$ iff $(\mathcal{A}, S) \models \neg(\neg F, \neg G)$.
$(\mathcal{A}, S) \models \neg F$ iff $(\mathcal{A}, S) =\!| F$.
$(\mathcal{A}, S) =\!| F$ iff $\neg F \in S$ where F is a ground atom.
$(\mathcal{A}, S) =\!| KF$ iff $(\mathcal{A}, S) \not\models KF$.
$(\mathcal{A}, S) =\!| MF$ iff $(\mathcal{A}, S) \not\models MF$.
$(\mathcal{A}, S) =\!| (F, G)$ iff $(\mathcal{A}, S) =\!| F$ or $(\mathcal{A}, S) =\!| G$.
$(\mathcal{A}, S) =\!| (F; G)$ iff $(\mathcal{A}, S) =\!| F$ and $(\mathcal{A}, S) =\!| G$.

An *epistemic logic program* or *ELP* is a finite set of ELP rules of the form:

$$F_1; F_2; \cdots; F_k \leftarrow G_1, \cdots, G_m, not\, G_{m+1}, \cdots, not\, G_n. \quad (1)$$

Here $F_1, \cdots, F_k$ are (objective) formulae, $G_1, \cdots, G_m$ are (objective) formulae or subjective formulaes, and $G_{m+1}, \cdots, G_n$ are (objective) formulae.

Note that ELP also allows nested expressions but in a restricted form.

For an epistemic logic program Π, its semantics is given by its *world view* which is defined in the following steps:
Step 1. Let Π be an epistemic logic program containing neither belief operators K and M nor negation as failure *not*. A set S of ground literals is called a *belief set* of Π iff S is

a minimal set of satisfying conditions: (i) for each rule $F_1; F_2; \cdots; F_k \leftarrow G_1, \cdots, G_m$ from Π such that $S \models (G_1, \cdots, G_m)$ we have $S \models (F_1; F_2; \cdots; F_t)$; and (ii) if S contains a pair of complementary literals, then S is the set Lit of all literals (called inconsistent belief set).

Step 2. Let Π be an epistemic logic program not containing modal operators K and M and S be a set of ground literals in the language of Π. By Π_S we denote the result of (i) removing from Π all the rules containing formulas of the form $not G$ such that $S \models G$ and (ii) removing from the rules in Π all other occurrences of formulas of the form $not G$. S is a *belief set* of Π iff S is a belief set of Π_S.

Step 3. Finally, let Π be an arbitrary epistemic logic program and $\mathcal{A}$ a collection of sets of ground literals in its language. By $\Pi_{\mathcal{A}}$ we denote the epistemic logic program obtained from Π by (i) removing from Π all rules containing formulas of the form G such that G is subjective and $\mathcal{A} \not\models G$, and (ii) removing from rules in Π all other occurrences of subjective formulas. S is a *belief set* of Π iff S is a belief set of $\Pi_{\mathcal{A}}$.

Example 5. The following epistemic logic program Π:

$$a \leftarrow$$
$$b; c \leftarrow$$
$$d \leftarrow Ka$$
$$e \leftarrow Mb$$

has a unique world view $\{\{a, b, d, e\}, \{a, c, d, e\}\}$.

Observe that the world view of the above program Π is also the equilibrium view of Π. This is not surprising because we will show that, for any epistemic logic program Π, $\mathcal{A}$ is a world view of Π if and only if $\mathcal{A}$ is an equilibrium view of Π.

The negation $\neg$ in epistemic program actually corresponds to the strong negation in extended logic program. In the rest of this section our discussion is temporarily restricted to epistemic programs that do not contain $\neg$. In Section 6, we will see that the results here are all valid for arbitrary epistemic programs.

We first introduce some notations. By viewing an epistemic logic program Π as an epistemic theory in epistemic HT-logic, $(\mathcal{A}, J, I) \models \Pi$ means that each rule is satisfied in the epistemic HT-interpretation $(\mathcal{A}, J, I)$. Under the world view semantics, on the other hand, we say that a rule of the form (1) *is satisfied* in a pair $(\mathcal{A}, S)$ if the fact $(\mathcal{A}, S) \models (G_1, \cdots, G_m)$, $(\mathcal{A}, S) \not\models G_{m+1}, \cdots, (\mathcal{A}, S) \not\models G_n$ implies $(\mathcal{A}, S) \models F_1; \cdots; F_k$. $(\mathcal{A}, S) \models \Pi$ means that each rule of Π is satisfied in $(\mathcal{A}, S)$. If Π does not contain any belief operators, we simply use $S \models \Pi$ to denote $(\mathcal{A}, S) \models \Pi$ (recall that the truth values of objective formulas are irrelevant to $\mathcal{A}$.

Lemma 2. *Let $(\mathcal{A}, I^H, I^T)$ be an epistemic HT-interpretation and Π an epistemic logic program without containing negation as failure. $(\mathcal{A}, J, I) \models \Pi$ if and only if $(\mathcal{A}, J) \models \Pi$.*

Lemma 3. *Let Π be an epistemic logic program and $(\mathcal{A}, J, I)$ be an epistemic HT-interpretation. Then $(\mathcal{A}, J, I) \models \Pi$ if and only if $J \models (\Pi_{\mathcal{A}})_I$. Here $(\Pi_{\mathcal{A}})_I$ is obtained through Step 3 and Step 2 in the definition of the world views.*

Proof. ($\Rightarrow$) Suppose $(\mathcal{A}, J, I) \models \Pi$, we want to show $J \models (\Pi_{\mathcal{A}})_I$.

If $R \in \Pi$, we can assume that R is of the form (1). If R satisfies $(\mathcal{A}, I) \models G_i$ for every i with $r+1 \leq i \leq m$ and $(\mathcal{A}, I) \not\models G_j$ for every i with $m+1 \leq j \leq n$, we use $(R_{\mathcal{A}})_I$ to denote the reduction of R with respect to $\mathcal{A}$ and I: $F_1; \cdots; F_k \leftarrow G_1, \cdots, G_r$. In general, $(R_{\mathcal{A}})_I$ may be undefined. Note that $(\Pi_{\mathcal{A}})_I = \{(R_{\mathcal{A}})_I \mid R \in \Pi\}$.

For any rule of $((\Pi_{\mathcal{A}})_I)$, it must be of the form $(R_{\mathcal{A}})_I$ for some rule $R \in \Pi$. So we need only to prove that $J \models (R_{\mathcal{A}})_I$ for $R \in \Pi$.

By the definition of the program reduction, we have the following two facts:

1. $(\mathcal{A}, I) \models G_i$ for $r+1 \leq i \leq m$ and
2. $(\mathcal{A}, I) \not\models G_i$ for $m+1 \leq i \leq n$.

Suppose $J \models body((R_{\mathcal{A}})_I))$, we want to show $J \models head((R_{\mathcal{A}})_I)$. That is, $J \models (F_1; \cdots; F_k)$. Since the body of $(R_{\mathcal{A}})_I$ is now the conjunction of $G_1, \cdots$, and G_r, we have $J \models G_i$ for all i with $1 \leq i \leq r$.

We prove that $(\mathcal{A}, J, I) \models body(R)$ by considering three different cases:

Case 1. Since $J \models G_i$ for $1 \leq i \leq r$, then $G_i \in J$ and thus $(\mathcal{A}, J, I, t) \models G_i$ for any $t \in \{H, T\}$.

Case 2. If $r+1 \leq i \leq m$, then G_i is a subjective literal and it is of form either KO_i' or MO_i' for some objective literal O_i. If $G_i = KO_i$, by $(\mathcal{A}, I) \models G_i$ for $r+1 \leq i \leq m$, $(\mathcal{A}, I) \models KO_i$. This means $(\mathcal{A}, I') \models O_i$ for all $I' \in \mathcal{A}$, which implies $(\mathcal{A}, I^H, I^T, t) \models O_i$ for all sets J^H, J^T of atoms with $J^H \subseteq J^T$. Thus $(\mathcal{A}, J, I, t) \models KO_i$ or $(\mathcal{A}, J, I, t) \models G_i$ for $r+1 \leq i \leq m$. Similarly, we also have $(\mathcal{A}, J, I, t) \models G_i$ for $r+1 \leq i \leq m$ if $G_i = MO_i$.

Case 3. If $(\mathcal{A}, I) \not\models G_i$ for $m+1 \leq i \leq n$, then $G_i \in I$. Since G_i is objective and $J \subseteq I$, $(\mathcal{A}, J, I) \not\models G_i$.

Combining Cases 1-3, we have $(\mathcal{A}, J, I) \models body(R)$. By $(\mathcal{A}, J, I) \models R$, we have $(\mathcal{A}, J, I) \models head(R)$. By $head(R) = head((R_{\mathcal{A}})_I)$ and thus $(\mathcal{A}, J, I) \models head((R_{\mathcal{A}})_I)$.

($\Leftarrow$) Suppose $J \models (\Pi_{\mathcal{A}})_I$, we show $(\mathcal{A}, J, I) \models \Pi$ by considering the following three cases:

For any rule $R \in \Pi$, assume R is of form (1),

Case 1. If $(\mathcal{A}, I) \not\models G_i$ for some i with $r+1 \leq i \leq m$, then $G_i \notin I$ and thus $G_i \notin J$. This implies $(\mathcal{A}, J, I) \not\models G_i$. In this case, $(\mathcal{A}, J, I) \not\models body(R)$. So $(\mathcal{A}, J, I) \models R$.

Case 2. If $(\mathcal{A}, I) \models G_i$ for some i with $m+1 \leq i \leq n$, then $G_i \in I$. This implies $(\mathcal{A}, J, I, T) \models G_i$. In this case, $(\mathcal{A}, J, I) \not\models body(R)$. So $(\mathcal{A}, J, I) \models R$.

Case 3. If neither Case 1 nor Case 2, then $(\mathcal{A}, J, I) \models G_{r+1} \wedge \cdots \wedge G_m \wedge \neg G_{m+1} \wedge \cdots \wedge \neg G_n$ and thus $(R_{\mathcal{A}})_I$ is well defined.
If $(\mathcal{A}, J, I) \models G_1 \wedge \cdots \wedge G_r$, then $J \models G_1 \wedge \cdots \wedge G_r$. Since $J \models (R_{\mathcal{A}})_I$, we have $J \models head((R_{\mathcal{A}})_I)$. That is, $J \models head(R)$.

Lemma 4. *Let Π be an ELP and $(\mathcal{A}, J, I)$ be a epistemic HT-interpretation. Then $(\mathcal{A}, I, I)$ is an equilibrium model of Π if and only if I is a belief set of $(\Pi_{\mathcal{A}})_I$.*

The above lemma, obtained directly from Lemma 3, implies the following result.

Theorem 2. *Let Π be an epistemic logic program and $\mathcal{A}$ be a collection of sets of atoms. Then $\mathcal{A}$ is a world view of Π if and only if $\mathcal{A}$ is an equilibrium view of Π.*

5 Strong Equivalence of Nested Epistemic Logic Programs

Recently, researchers have addressed the problem of characterizing the strong equivalence of logic programs under the answer sets. In particular, the result in [7] shows that the strong equivalence of nested logic programs can be characterized in term of the equivalence of formulas in monotonic logic (the HT-logic). Here we are interested in extending this result to NELPs under the equilibrium view semantics. We say two NELPs Π_1 and Π_2 are *equivalent* if they have the same equilibrium views. A NELP Π_1 is said to be *strong equivalent* to another NELP Π_2 if, for every NELP Π, $\Pi_1 \cup \Pi$ and $\Pi_2 \cup \Pi$ are equivalent. It is well-known that equivalence of two programs does not implies their strong equivalence in general (see [2,8,7] for more examples of strongly equivalent programs).

Similarly, two theories Π_1 and Π_2 in the epistemic HT-logic is *equivalent* if they have the same set of models.

Theorem 3. *For any nested epistemic logic programs Π_1 and Π_2, the following conditions are equivalent:*

(1) *Π_1 is strongly equivalent to Π_2.*
(2) *Π_1 is equivalent to Π_2 in the epistemic HT-logic.*

By Theorem 2, it is easy to prove Theorem 3 since we have the following lemma, whose proof is similar to that of Theorem 1 in [7].

Lemma 5. *For any epistemic theories Γ_1 and Γ_2, the following conditions are equivalent:*

(1) *for every epistemic theory Γ, $\Gamma_1 \cup \Gamma$ and $\Gamma_2 \cup \Gamma$ have the same equilibrium models.*
(2) *Γ_1 is equivalent to Γ_2 in the epistemic HT-logic.*

For any two objective theories Γ_1 and Γ_2, they are equivalent in the logic of here-and-there if and only if they are equivalent in the epistemic HT-logic. Thus, by Theorem 2, the main result (Theorem 1) in [7] is a corollary of our Theorem 3:

Corollary 1. *For any nested logic programs Π_1 and Π_2, the following conditions are equivalent:*

(1) *Π_1 is strongly equivalent to Π_2.*
(2) *Π_1 is equivalent to Π_2 in the logic of here-and-there.*

By Theorem 2, the strong equivalence of epistemic logic programs under the world view semantics can also be verified by checking the equivalence of formulas in the epistemic HT-logic which is a monotonic logic.

Corollary 2. *For any epistemic logic programs Π_1 and Π_2, the following conditions are equivalent:*

(1) *Π_1 is strongly equivalent to Π_2.*
(2) *Π_1 is equivalent to Π_2 in the epistemic HT-logic.*

6 Adding Strong Negation in NELPs

In answer set programming, the syntax of logic programs usually allows both negation as failure and strong negation [5]. The second negation is denoted by $\neg$. It is well-known that this extension is very useful for representing and reasoning about incomplete information. In this section, we show how to add the second negation in NELPs. This can be done by an easy generalization of Section 5 in [7]. Technically, it is not hard to add the strong negation in the syntax of logic programs.

A *literal* is an atom a or its strong negation $\neg a$. By allowing arbitrary literals in place of atoms, *extended NELP formula*, *extended NELP rule* and *extended NELP* can be defined in the same way as we defined NELP formula, NELP rule and NELP. Following [5], the semantic of an extended NELP can be defined through a simple syntactic translation.

Given an extended NELP Π, we introduce a new symbol a' for each atom a in Π. Then Π can be translated into a NELP Π' by replacing each negative literal $\neg a$ with a'. Note that Π' does not contain the strong negation. For any expression E, we use E' to denote the expression obtained by replacing every $\neg a$ with a'. Denote $Cons(\Pi) = \{\bot \leftarrow a, a' \mid a \text{ is literal in } \Pi\}$. Then we say $\mathcal{A}$ is an equilibrium view of Π if $\mathcal{A}'$ is an equilibrium view of $\Pi' \cup Cons(\Pi)$.

For nested logic programs (with strong negation) Π, a set X of literals is an answer set of Π iff X is X' is an answer set of $\Pi' \cup Cons(\Pi)$. Therefore, Theorem 1 is also true for nested logic programs with strong negation.

In the same way as in [5], the negation $\neg$ in an epistemic logic program can be eliminated by introducing new atom a' for each atom a. Thus, Theorem 2 is also true for epistemic logic program with strong negation.

Theorem 3 can also be generalized to extended NELPs.

Theorem 4. *For any extended NELPs Π_1 and Π_2, the following conditions are equivalent:*

(1) *Π_1 is strongly equivalent to Π_2.*
(2) *$\Pi_1 \cup Cons(\Pi_1)$ is equivalent to $\Pi_2 \cup Cons(\Pi_2)$ in the epistemic HT-logic.*

Proof. Π_1 is strongly equivalent to Π_2
if and only if
$\Pi_1' \cup Cons(\Pi_1)$ is strongly equivalent to $\Pi_2 \cup Cons(\Pi_2)$
if and only if
$\Pi_1 \cup Cons(\Pi_1)$ is equivalent to $\Pi_2 \cup Cons(\Pi_2)$ in the epistemic HT-logic.

7 Conclusions

In this paper, we introduced the epistemic HT-logic and based on this logic further developed a new type of logic programs called nested epistemic logic programs (NELPs). We showed that the equilibrium view semantics of NELPs generalizes the answer set semantics of nested logic programs as well as the world view semantics of epistemic logic programs. We also characterize the strong equivalence property of NELPs in terms of epistemic HT-logic.

Some important issues related to NELPs should be further investigated. Firstly, it is important to understand the computational properties of NELPs in detail from both theoretical and practical viewpoints. Secondly, as epistemic logic programs may be viewed as an effective formalism for representing and reasoning about agent's dynamic epistemic state, e.g. [12], it would be interesting to explore how this work can be improved by applying NELPs.

Acknowledgements. The authors would like to thank the three referees for their comments. In particular, the comments from one referee greatly helped the improvement of the paper.

References

1. C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.
2. E. Erdem and V. Lifschitz. Transformations of logic programs related to causality and planning. In *Proceedings of the 5th International Conference on Logic Programming and Nonmonotonic Reasoning (LNAI 1730)*, pages 107–116, 1999.
3. M. Gelfond. Logic programming and reasoning with incomplete information. *Annals of Mathematics and Artificial Intelligence*, 12:98–116, 1994.
4. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proceedings of the International Conference on Logic Programming*, pages 1070–1080. The MIT Press, 1988.
5. M. Gelfond and V. Lifschitz. Logic programs with classical negation. In *Proceedings of the International Conference on Logic Programming*, pages 579–597, 1990.
6. V. Lifschitz. Answer set planning. In *Proceedings of the International Conference on Logic Programming*, pages 23–37. The MIT Press, 1999.
7. V. Lifschitz, D. Pearce, and A. Valverde. Strongly equivalent logic programs. *ACM Transactions on Computationl Logic*, 2(4):426–541, 2001.
8. V. Lifschitz, L. Tang, and H. Turner. Nested expressions in logic programs. *Annals of Mathematics and Artificial Intelligence*, 25:369–389, 1999.
9. F. Lin and Y. Shoham. A logic of knowledge and justified assumptions. *Artificial Intelligence*, 57:271–289, 1992.
10. D. Pearce. From here to there: stable negation in logic programming. In D. Gabbay and H. Wansing, editors, *What is Negation?* Springer-Verlag, 1997.
11. D. Pearce. A new logical characterization of stable models and answer sets. In J. Dix, L. Pereira, and T. Przymusinski, editors, *Non-Monotonic Extensions of Logic Programming (LNAI 1216)*, pages 57–70. Springer-Verlag, 1997.
12. Y. Zhang. Minimal change and maximal coherence for epistemic logic program updates. In *Proceedings of IJCAI-03*, pages 112–117. 2003.

An Algebraic Account of Modularity in ID-Logic*

Joost Vennekens and Marc Denecker

Dept. of Computer Science, K.U.Leuven
Celestijnenlaan 200A
B-3001 Leuven, Belgium
{joost.vennekens, marc.denecker}@cs.kuleuven.ac.be

Abstract. ID-logic uses ideas from the field of logic programming to extend second order logic with non-monotone inductive defintions. In this work, we reformulate the semantics of this logic in terms of approximation theory, an algebraic theory which generalizes the semantics of several non-monotonic reasoning formalisms. This allows us to apply certain abstract modularity theorems, developed within the framework of approximation theory, to ID-logic. As such, we are able to offer elegant and simple proofs of generalizations of known theorems, as well as some new results.

1 Introduction

Inductive definitions are common in mathematical practice. For instance, the non-monotone inductive definition of the satisfaction relation $\models$ (see Definition 1 in Section 2.2) can be found in most textbooks on first-order logic. This prevalence of inductive definitions indicates that these offer a natural and well-understood way of representing knowledge. At the same time, inductive definitions cannot easily be expressed in classical logic. For instance, the transitive closure of a graph is one of the simplest concepts typically defined by induction—such a definition might consist of the following two rules: if (x, y) is an edge of the graph, (x, y) belongs to the transitive closure and if $\exists z$ such that both (x, z) and (z, y) belong to the transitive closure, then (x, y) belongs to the transitive closure—yet it can be shown that this concept cannot be defined in first-order logic. While second-order logic does allow the representation of such simple definitions, the resulting formula might not always be very natural and the use of second-order logic itself may be undesirable, e.g., due to computational considerations. Moreover, even this methodology breaks down when faced with non-monotone inductive definitions, such as that of the satisfaction relation.

It turns out, however, that certain knowledge representation logics do allow even non-monotone inductive definitions to be correctly formalized in an intuitive way. Particularly suited for this are logic programs under the well-founded

* This work is supported by FWO-Vlaanderen, European Framework 5 Project WASP, and by GOA/2003/08.

C. Baral et al. (Eds.): LPNMR 2005, LNAI 3662, pp. 291–303, 2005.
© Springer-Verlag Berlin Heidelberg 2005

model semantics. In fact, one could even go so far as to explain the semantical foundations of this logic themselves as precisely a formalization of the principle of inductive definition [Den01]. The language of *ID-logic* uses the well-founded semantics to extend classical logic with a new "inductive definition" primitive. In the resulting formalism, all kinds of definitions regularly found in mathematical practice—e.g., monotone inductive definitions, non-monotone inductive definitions over a well-ordered set, and iterated inductive definitions—can be represented in a uniform way. Moreover, this representation neatly corresponds to the form such a definitions would take in a mathematical text. For instance, in ID-logic the transitive closure of a graph can be defined as:

$$\left\{\begin{array}{l}\forall x, y\ TransCl(x,y) \leftarrow Edge(x,y).\\ \forall x, y\ TransCl(x,y) \leftarrow (\exists z\ TransCl(x,z) \wedge TransCl(z,y)).\end{array}\right\}$$

However, ID-logic is able to handle more than only mathematical concepts. Indeed, inductive definitions are also useful in common-sense reasoning. For instance, in [DT04a], it was shown that situation calculus can be given a natural representation as an iterated inductive definition. The resulting theory is able to correctly handle tricky issues such as recursive ramifications, and is in fact, to the best of our knowledge, the most general representation of this calculus to date. In general, definitions are a distinctive and important form of human expert knowledge; as a uniform and natural way of representing this kind of knowledge, ID-logic provides a useful contribution to the field of knowledge representation.

The goal of this paper is to study modularity properties for ID-logic. Modularity properties deal with the relation between a theory and its components. Typical examples are so-called splitting results, which allow large theories to be rewritten as equivalent sets of sub-theories. Such properties are of interest, because they may offer additional insight into the semantics of a formalism, can be used to guarantee that certain transformations are equivalence preserving, or may lead allow more efficient computations.

Modularity properties have been studied for a large number of different formalisms. Recently, an algebraic theory of modularity [VGD04b, VGD04a] was developed within the framework of *approximation theory*, a general fixpoint theory for arbitrary operators, which naturally captures the semantics of logic programs, auto-epistemic logic, and default logic [DMT03, DMT00]. These abstract results have since been used to unify several concrete splitting theorems: [VGD04a] generalizes results concerning autoepistemic logic [GP92], and [VGD04b] (partially) generalizes results for logic programming [LT94].

Here, we apply this algebraic modularity theory to ID-logic. First, we show how the semantics of this logic can be reformulated in terms of approximation theory. By doing so, we are able to apply the aforementioned splitting theorems (and a small extension thereof) to ID-logic and obtain a generalization of results from [DT04b], as well as some new results.

The structure of this paper is as follows. Section 2 introduces approximation theory and ID-logic. Section 3 summarizes the algebraic modularity results which will be used. In Section 4, we then apply those results to ID-logic.

2 Preliminaries

2.1 Approximation Theory

Approximation theory is a general fixpoint theory for arbitrary operators. Our presentation of this theory is based on [DMT00, DMT03].

Let $\langle L, \leq \rangle$ be a lattice. An element (x, y) of the square L^2 of the domain of such a lattice, can be seen as denoting an interval $[x, y] = \{z \in L \mid x \leq z \leq y\}$. Using this intuition, we can derive a *precision* order $\leq_p$ on the set L^2 from the order $\leq$ on L: for each $x, y, x', y' \in L, (x, y) \leq_p (x', y')$ iff $x \leq x'$ and $y' \leq y$. Indeed, if $(x, y) \leq_p (x', y')$, then $[x, y] \supseteq [x', y']$. It can easily be shown that $\langle L^2, \leq_p \rangle$ is also a lattice, which is called the *bilattice* corresponding to L. Moreover, if L is complete, then so is L^2. As an interval $[x, x]$ contains precisely one element, namely x itself, elements (x, x) of L^2 are called *exact*. The set of all exact elements of L^2 forms a natural embedding of L in L^2. A pair (x, y) only corresponds to a non-empty interval if $x \leq y$. Such pairs are called *consistent*.

Approximation theory is based on the study of operators on bilattices L^2 which are monotone w.r.t. the precision order $\leq_p$. Such operators are called *approximations*. For an approximation A and $x, y \in L$, we denote by $A^1(x, y)$ and $A^2(x, y)$ the unique elements of L, for which $A(x, y) = (A^1(x, y), A^2(x, y))$. An approximation *approximates* an operator O on L if for each $x \in L$, $A(x, x)$ contains $O(x)$, i.e. $A^1(x, x) \leq O(x) \leq A^2(x, x)$. An approximation is *symmetric* if for each pair $(x, y) \in L^2$, if $A(x, y) = (x', y')$ then $A(y, x) = (y', x')$.

For an approximation A on L^2, the following two operators on L can be defined: the function $A^1(\cdot, y)$ maps an element $x \in L$ to $A^1(x, y)$, i.e. $A^1(\cdot, y) = \lambda x.A^1(x, y)$, and the function $A^2(x, \cdot)$ maps an element $y \in L$ to $A^2(x, y)$, i.e. $A^2(x, \cdot) = \lambda y.A^2(x, y)$. As all such operators are monotone, they all have a unique least fixpoint. We define an operator $C_A^{\downarrow}$ on L, which maps each $y \in L$ to $lfp(A^1(\cdot, y))$ and, similarly, an operator $C_A^{\uparrow}$, which maps each $x \in L$ to $lfp(A^2(x, \cdot))$. $C_A^{\downarrow}$ is called the *lower stable operator* of A, while $C_A^{\uparrow}$ is the *upper stable operator* of A. Both these operators are anti-monotone. Combining these two operators, the operator $\mathcal{C}_A$ on L^2 maps each pair (x, y) to $(C_A^{\downarrow}(y), C_A^{\uparrow}(x))$. This operator is called the *partial stable operator* of A. Because the lower and upper partial stable operators $C_A^{\downarrow}$ and $C_A^{\uparrow}$ are anti-monotone, the partial stable operator $\mathcal{C}_A$ is monotone. If an approximation A is symmetric, its lower and upper partial stable operators will always be equal, i.e. $C_A^{\downarrow} = C_A^{\uparrow}$.

An approximation A defines a number of different fixpoints: the least fixpoint of an approximation A is called its *Kripke-Kleene fixpoint*, fixpoints of its partial stable operator $\mathcal{C}_A$ are *stable fixpoints* and the least fixpoint of $\mathcal{C}_A$ is called the *well-founded fixpoint* of A. As shown in [DMT00, DMT03], these fixpoints correspond to various semantics of logic programming, auto-epistemic logic and default logic.

Finally, it should also be noted that the concept of an approximation as defined in [DMT00] corresponds to our definition of a *symmetric* approximation.

2.2 ID-Logic

ID-logic [DT04b, DT04a] extends second-order logic with non-monotone inductive definitions. Before defining this logic in its entirity, we first introduce basic second order logic. Following [DT04a], we do this in a slightly non-standard way. In particular, no distinction is made between constant symbols and variables.

We assume an infinite supply of *object symbols* $x, y, \ldots$, *function symbols* $f/n, g/n, \ldots$ of every arity n, and *predicate symbols* $P/n, Q/n, \ldots$ of every arity n. A *vocabulary* Σ is a set of symbols. We denote by Σ_o the object symbols in Σ, by Σ_f the function symbols, and by Σ_P the predicate symbols. *Terms* and *atoms* of Σ are defined in the usual way. A *formula of* Σ is inductively defined as:

- a Σ-atom $P(t_1, \ldots, t_n)$ is a Σ-formula;
- if ϕ is a Σ-formula, then so is $\neg\phi$;
- if ϕ_1 and ϕ_2 are Σ-formulas, then so is $(\phi_1 \vee \phi_2)$;
- if ϕ is a $(\Sigma \cup \{\sigma\})$-formula and σ an (object, function or predicate) symbol, then $(\exists\sigma\ \phi)$ is a Σ-formula.

If in all quantifications $\exists\sigma$ of a formula ϕ, σ is an object symbol, ϕ is called *first order.*

Given a certain domain D, a symbol σ can be assigned a *value* in D:

- if $\sigma \in \Sigma_o$, a value for σ in D is an element of D;
- if $\sigma/n \in \Sigma_f$, a value for σ in D is a function of arity n in D;
- if $\sigma/n \in \Sigma_P$, a value for σ in D is a relation of arity n in D.

A *structure* S *for vocabulary* Σ, or Σ*-structure* S, consists of a domain, denoted S_D, and a mapping from each symbol σ in Σ to a value σ^S in S_D for σ. A vocubalary Σ is a *sub-vocabulary* of Σ' iff $\Sigma \subseteq \Sigma'$. The *restriction* $S'|_\Sigma$ of a Σ'-structure S' to a sub-vocabulary Σ, is the Σ-structure S for which $S_D = S'_D$ and, for each symbol σ of Σ, $\sigma^S = \sigma^{S'}$. Under the same conditions, S' is called an *extension* of S to Σ'. The set of all structures extending S to Σ' is denoted by $\mathcal{S}^S_{\Sigma'}$. For each value a in S_D for a symbol σ, we denote by $S[\sigma/a]$ the extension S' of S to $\Sigma \cup \{\sigma\}$, such that $\sigma^{S'} = a$.We also extend this notation to tuples $\mathbf{x}$ and $\mathbf{a}$, and to pairs (X, Y) of Σ-structures sharing the same domain, i.e., $(X, Y)[\mathbf{x}/\mathbf{a}] = (X[\mathbf{x}/\mathbf{a}], Y[\mathbf{x}/\mathbf{a}])$.

The *value* of a Σ-term t in a Σ-structure S, also denoted t^S, is inductively defined as: $(f(t_1, \ldots, t_n))^S = f^S(t_1^S, \ldots, t_n^S)$, for a function symbol f and terms $t_1, \ldots, t_n$. We now define a satisfaction relation between structures and formulas:

Definition 1. *For a* Σ*-structure* S *and* Σ*-formula* ϕ*, the relation "*S *satisfies* ϕ*", denoted* $S \models \phi$*, is inductively defined as:*

- $S \models P(\mathbf{t})$ *iff* $\mathbf{t}^S \in P^S$*;*
- $S \models (\phi_1 \vee \phi_2)$ *iff* $S \models \phi_1$ *or* $S \models \phi_2$*;*
- $S \models \neg\phi$ *iff* $S \not\models \phi$*;*
- $S \models (\exists\sigma\ \phi)$ *iff there exists a value* a *for* σ *in the domain* S_D*, such that* $S[\sigma/a] \models \phi$*;*

A *pre-interpretation* H *for* Σ is a structure for the language $\Sigma_o \cup \Sigma_f$, i.e., one which interprets only the object and function symbols of Σ. A structure S extending H to Σ is called an H*-interpretation*. Clearly, H-interpretations can only differ in their assignment of relations (over the common domain S_H) to predicate symbols. Given a domain D, a *domain atom* is a pair $(P, \mathbf{a})$, with P/n a predicate of Σ and $\mathbf{a} \in D^n$. We also write such a pair as $P(\mathbf{a})$. The function At_H is defined as mapping an H-interpretation S to the set of all domain atoms $P(\mathbf{a})$ in H_D, for which $\mathbf{a} \in P^S$. At_H is a one-to-one correspondence between H-interpretations and sets of domain atoms for H_D. The set of all H-interpretations is a complete lattice w.r.t. to the truth order $\leq_t$, defined as: $S \leq_t S'$ iff $At_H(S) \subseteq At_H(S')$ (or, equivalently, for each predicate P, $P^S \subseteq P^{S'}$).

Next, we explain how this logic can be extended with inductive definitions. We do this using concepts from approximation theory. In this, our presentation differs from the more direct approach taken in [DT04a].

As a first step, we extend the notion of satisfation to pairs (X, Y) of structures.

Definition 2. *Let* H *be a pre-interpretation for* Σ*,* X *and* Y H*-interpretations, and* ϕ *a* Σ*-formula. The relation "*(X, Y) satisfies ϕ*", denoted* $(X, Y) \models \phi$ *is inductively defined by:*

- $(X, Y) \models P(\mathbf{t})$ *iff* $\mathbf{t}^H \in P^X$*;*
- $(X, Y) \models (\phi_1 \vee \phi_2)$ *iff* $(X, Y) \models \phi_1$ *or* $(X, Y) \models \phi_2$*;*
- $(X, Y) \models \neg\phi$ *iff* $(Y, X) \not\models \phi$*;*
- $(X, Y) \models (\exists\sigma\ \phi)$ *iff there exists a value* a *for* σ *in* H_D*, such that* $(X, Y)[\sigma/a] \models \phi$*;*

Observe that in the rule for $\neg\phi$, the roles of X and Y are switched. This causes all positively occurring atoms in ϕ to be evaluated in X, while all negatively occurring atoms in ϕ are evaluated in Y. To motivate this definition, let us consider a structure S approximated by (X, Y), i.e. such that $X \leq_t S \leq_t Y$. In the evaluation of ϕ in (X, Y), all positively occurring atoms are evaluated with respect to the underestimate X of S, and all negatively occurring atoms are evaluated with respect to the overestimate Y of S. Therefore, the truth value of ϕ in (X, Y) is an underestimate of the value of ϕ in S. Vice versa, in the evaluation of ϕ in (Y, X), all positively occurring atoms are evaluated in the overestimate Y while all negatively occurring atoms are evaluated in the underestimate X, and hence, the truth value of ϕ in (Y, X) is an overestimate of the value of ϕ in S.

Considering satisfaction in pairs of structures rather than single structures, corresponds to switching to a four-valued logic: ϕ is *true* according to (X, Y) if $(X, Y) \models \phi$ and $(Y, X) \models \phi$, *false* if $(X, Y) \not\models \phi$ and $(Y, X) \not\models \phi$, *unkown* if $(X, Y) \not\models \phi$ and $(Y, X) \models \phi$, and *inconsistent* if $(X, Y) \models \phi$ and $(Y, X) \not\models \phi$.

We now define the ID-logic syntax used for inductive definitions. Let Σ be a vocabulary. A *definitional rule* r of Σ is a formula $\forall\mathbf{x}\ A \leftarrow \phi$, with A a Σ-atom and ϕ a first-order $(\Sigma \cup \mathbf{x})$-formula. The atom A is called the *head,* $head(r)$*, of* r and ϕ is called the *body,* $body(r)$*, of* r. Note that the symbol "$\leftarrow$" in such a rule should not be read as material implication, but rather as a new language

primitive: the *definitional implication*. A rule r is said to be a *defining rule* of a predicate P if P is the predicate of $head(r)$. A *Σ-definition* Δ is a set of definitional rules. A predicate symbol having at least one defining rule r in Δ, is called a *defined predicate of* Δ. The set of all such predicates is denoted by $\mathcal{P}^d_\Delta$. Predicates of Σ_P which are not defined by Δ are *open in* Δ and the set of all such predicates is denoted by $\mathcal{P}^o_\Delta$. The notations Σ^o_Δ and Σ^d_Δ are used to denote the vocabularies $\Sigma_o \cup \Sigma_f \cup \mathcal{P}^o_\Delta$ and $\Sigma_o \cup \Sigma_f \cup \mathcal{P}^d_\Delta$, respectively.

Using this syntax, the well-known simultaneous inductive definition of the even and odd numbers (i.e., 0 is an even number, each successor of an even number is an odd number, and vice versa) can be written as:

Example 1.

$$\Delta_{even} = \left\{ \begin{array}{c} Even(0). \\ \forall x\ Even(s(x)) \leftarrow Odd(x). \\ \forall x\ Odd(s(x)) \leftarrow Even(x). \end{array} \right\}$$

Intuitively, such an inductive definition describes a process by which, given some fixed interpretation of the open predicates, new elements of the defined relations can be derived from a set of already known elements. The formal definition of the semantics of ID-logic captures this intuition, by associating a class of operators to a definition Δ. More precisely, for each interpretation O of the open predicates of Δ, an operator $\mathcal{T}^O_\Delta$ is defined, which maps an estimate (X,Y) of the defined relations to a more precise estimate $\mathcal{T}^O_\Delta(X,Y) = (X',Y')$. The new lower bound X' is constructed by underestimating the truth of the bodies of the rules in Δ, i.e., by evaluating these in (X,Y). When constructing the new upper bound Y', on the other hand, the truth of the bodies of these rules is overestimated, i.e., evaluated in (Y,X).

Definition 3. *Let Δ be a Σ-definition and O a Σ^o_Δ-structure. We define a function U^O_Δ from the bilattice $(\mathcal{S}^O_\Sigma)^2$ to $\mathcal{S}^O_\Sigma$ as $U^O_\Delta(X,Y) = S$, with for each $P \in \Sigma^d_\Delta$: $\mathbf{a} \in P^S$ iff there exists a rule $(\forall \mathbf{x}\ P(\mathbf{t}) \leftarrow \phi)$ in Δ and a value $\mathbf{c}$ for $\mathbf{x}$, such that $(X,Y)[\mathbf{x}/\mathbf{c}] \models \phi$ and $\mathbf{a} = \mathbf{t}^{S[\mathbf{x}/\mathbf{c}]}$. The operator $\mathcal{T}^O_\Delta$ on $(\mathcal{S}^O_\Sigma)^2$ is defined as, for all $X,Y \in \mathcal{S}^O_\Sigma$:*

$$\mathcal{T}^O_\Delta(X,Y) = (U^O_\Delta(X,Y), U^O_\Delta(Y,X)).$$

If an estimate (X,Y) is more precise than an estimate (X',Y'), i.e., $X' \leq_t X$ and $Y \leq_t Y'$, then $\mathcal{T}^O_\Delta(X,Y)$ will also be more precise than $\mathcal{T}^O_\Delta(X',Y')$. In other words, each operator $\mathcal{T}^O_\Delta$ is an approximation. As such, each $\mathcal{T}^O_\Delta$ has a well-founded fixpoint. We now use this to define the semantics of the logic.

Definition 4. *Let Σ be a vocabulary. An* ID-logic formula *is inductively defined by extending the definition of a formula with the additional base case:*

- *A definition Δ is an ID-logic formula.*

The corresponding base case for the satisfaction relation is:

- *$S \models \Delta$ iff $X|_{\Sigma^d_\Delta} = S|_{\Sigma^d_\Delta} = Y|_{\Sigma^d_\Delta}$, with (X,Y) the well-founded fixpoint of $\mathcal{T}^O_\Delta$, with $O = S|_{\Sigma^o_\Delta}$.*

Note that, even though this definition uses the operator $\mathcal{T}_\Delta^S$ on pairs of structures, the eventual models of a definition are always single structures S. The intuition here is that a definition should completely define its defined predicates, i.e., there should be no tuples for which it is "unknown" whether they belong to the defined relations or not.

Definition 5. *Let Σ be a vocabulary. A Σ-definition Δ is* total *in a Σ_Δ^o -structure O iff $X = Y$, with (X, Y) the well-founded fixpoint of $\mathcal{T}_\Delta^O$.*

3 Algebraic Splitting Results

In this section, we summarize and extend results from [VGD04b]. First, we introduce some basic definitions and notations. Let I be a set, which we call the *index set*, and for each $i \in I$, let S_i be a set. The *product set* $\bigotimes_{i \in I} S_i$ is the following set of functions:

$$\bigotimes_{i \in I} S_i = \{f \mid f : I \to \bigcup_{i \in I} S_i \text{ such that } \forall i \in I : f(i) \in S_i\}.$$

If, for instance, I is $\{1, \ldots, n\}$, the product $\bigotimes_{i \in I} S_i$ is (isomorphic to) the cartesian product $S_1 \times \cdots \times S_n$.

If each S_i is partially ordered by some $\leq_i$, this induces the *product order* $\leq_\otimes$ on $\otimes_{i \in I} S_i$: $\forall x, y \in \otimes_{i \in I} S_i$, $x \leq_\otimes y$ iff $\forall i \in I : x(i) \leq_i y(i)$. It can easily be shown that if all $\langle S_i, \leq_i \rangle$ are (complete) lattices, then $\langle \otimes_{i \in I} S_i, \leq_\otimes \rangle$ is also a (complete) lattice; this is the *product lattice* of the lattices S_i.

From now on, we only consider product lattices with a *well-founded* index set, i.e., index sets I with a partial order $\preceq$ such that each non-empty subset of I has a $\preceq$-minimal element. This allows us to use inductive arguments in dealing with elements of product lattices.

The following notations are used. Let L be a product lattice $\otimes_{i \in I} L_i$. For $x \in L$ and $i \in I$, we abbreviate the restriction $x|_{\{j \in I | j \preceq i\}}$ by $x|_{\preceq i}$. We also use similar abbreviations $x|_{\prec i}$, $x|_i$ and $x|_{\not\preceq i}$. If i is a minimal element of the well-founded set I, $x|_{\prec i}$ is defined as the empty function. For any subset J of I, the set $\{x|_J \mid x \in L\}$, ordered by the appropriate restriction $\leq_\otimes|_J$ of the product order, is also a lattice. This sublattice of L is of course equal to the product lattice $\otimes_{j \in J} L_j$. If J is of the form $\{j \in I \mid j \preceq i\}$ for some i, we simply write $L|_{\preceq i}$ for $L|_J$. Similarly, $L|_{\prec i}$ is written for $\otimes_{j \prec i} L_i$.

If f, g are functions $f : A \to B$, $g : C \to D$ and the domains A and C are disjoint, we denote by $f \sqcup g$ the function from $A \cup C$ to $B \cup D$, such that for all $a \in A$, $(f \sqcup g)(a) = f(a)$ and for all $c \in C$, $(f \sqcup g)(c) = g(c)$. Furthermore, for any g whose domain is disjoint from the domain of f, we call $f \sqcup g$ an *extension* of f. For each element x of a product lattice L and each index $i \in I$, the extension $x|_{\prec i} \sqcup x|_i$ of $x|_{\prec i}$ is clearly equal to $x|_{\preceq i}$. To ease notation, we sometimes write $x(i)$ instead of $x|_i$ in such expressions, i.e. we identify an element a of the ith lattice L_i with the function from $\{i\}$ to L_i which maps i to a. Similarly, $x|_{\prec i} \sqcup x(i) \sqcup x|_{\not\preceq i} = x$.

Our goal is now to study operators on product lattices. Let $\langle I, \preceq \rangle$ be a well-founded index set and let $L = \otimes_{i \in I} L_i$ be a product lattice. Intuitively, an operator O on L is stratifiable over $\preceq$, if the value $(O(x))(i)$ of $O(x)$ in the ith level only depends on values $x(j)$ for which $j \preceq i$.

Definition 6. *An operator O on a product lattice L is* stratifiable *iff $\forall x, y \in L, \forall i \in I$: if $x|_{\preceq i} = y|_{\preceq i}$ then $O(x)|_{\preceq i} = O(y)|_{\preceq i}$.*

It is possible to characterize stratifiablity in a more constructive manner. The following proposition shows that stratifiablity of an operator O on a product lattice L is equivalent to the existence of a family of operators on each lattice L_i (one for each partial element u of $L|_{\prec i}$), which mimics the behaviour of O on this lattice.

Proposition 1. *Let O be an operator on a product lattice L. O is stratifiable iff for each $i \in I$ and $u \in L|_{\prec i}$ there exists a unique operator O_i^u on L_i, such that for all $x \in L$:*

$$\textit{If } x|_{\prec i} = u \textit{ then } (O(x))(i) = O_i^u(x(i)).$$

The operators O_i^u are called the *components* of O. The main results of [VGD04b] are the following correspondences between various kinds of fixpoints of the original operator O and those of its components O_i^u:

Theorem 1. *Let L be a product lattice $\otimes_{i \in I} L_i$.*

- *If O is a stratifiable operator on L, then for each $x \in L$: x is a fixpoint of O iff $\forall i \in I : x(i)$ is a fixpoint of $O_i^{x|_{\prec i}}$.*
- *If O is a monotone stratifiable operator on L, then for each $x \in L$: x is the least fixpoint of O iff $\forall i \in I : x(i)$ is the least fixpoint of $O_i^{x|_{\prec i}}$.*
- *If O is a stratifiable approximation on the bilattice L^2, then for each $x \in L^2$: x is a stable (well-founded) fixpoint of O iff $\forall i \in I : x(i)$ is a stable (well-founded, respectively) fixpoint of $O_i^{x|_{\prec i}}$.*

This theorem allows us to incrementally construct any kind of fixpoint of a stratifiable operator, by constructing the corresponding fixpoints of its components in a bottom-up manner w.r.t. the well-founded order $\preceq$ on the index set.

We now extend this material from [VGD04b] with some additional results. More specifically, we not only want to split a stratifiable operators into its components, but also into sets of "bigger" operators, i.e., operators which may encompass several levels. For a subset J of I and $x \in L|_{I \setminus J}$, we denote by O_J^x the operator on $L|_J$ which maps each $y \in L|_J$ to $O(x \sqcup y)|_J$. Such operators O_J^x are called *recombinations of* O. Our goal is now to show that, for each partition $\mathcal{J}$ of I, a stratifiable operator O can be split into the recombinations O_J^x, with $J \in \mathcal{J}$. We do this, by showing that a recombination O_J^x is also stratifiable and can be split into the components of O itself.

Proposition 2. *Let O be a stratifiable operator. For each $J \subseteq I$ and $x \in L|_{I \setminus J}$, O_J^x is stratifiable.*

Proof. Let O^x_J be as above, $i \in J$, and $y, y' \in L|_J$, such that $y|_{\preceq i} = y'|_{\preceq i}$. By definition, $O^x_J(y) = O(x \sqcup y)|_J$. Because $(x \sqcup y)|_{\preceq i} = (x \sqcup y')|_{\preceq i}$, we have that, by stratifiability of O, $O^x_J(y)|_{\preceq i} = O(x \sqcup y)|_{\{j \in J | j \preceq i\}} = O(x \sqcup y')|_{\{j \in J | j \preceq i\}} = O^x_J(y')|_{\preceq i}$.

Proposition 3. *Let O be a stratifiable operator. For each $J \subseteq I$, $x \in L|_{I \setminus J}$, $i \in J$, and $u \in L|_{\{j \in J | j \prec i\}}$, the component $(O^x_J)^u_i$ of O^x_J equals the component $O_i^{u \sqcup (x|_{\prec i})}$ of O.*

Proof. Let $(O^x_J)^u_i$ be as above and let $y \in L_i$. By definition, for any z extending $u \sqcup y$ to J, $(O^x_J)^u_i(y) = O(x \sqcup z)|_i = (O_i^{(x \sqcup z)|_{\prec i}}(z|_i))|_i = O_i^{x|_{\prec i} \sqcup u}(y)$.

These two propositions now imply the wanted result.

Theorem 2. *Let O be a stratifiable operator and let $\mathcal{J}$ be a partition of I. Then, for each $x \in L$, x is a fixpoint (least fixpoint, stable fixpoint, or well-founded fixpoint) of O (assuming that O is monotone or an approximation, where appropriate) iff for each $J \in \mathcal{J}$, $x|_J$ is a fixpoint (least fixpoint, stable fixpoint, or well-founded fixpoint) of $O_J^{x|_{I \setminus J}}$.*

Proof. We only show the correspondence between fixpoints; the proofs of the other correspondences are similar. Let x be a fixpoint of O. By Theorem 1, this is equivalent to: $\forall i \in I$, $x|_i$ is a fixpoint of $O_i^{x|_{\prec i}}$. Because $\mathcal{J}$ partitions I, this is equivalent to $\forall J \in \mathcal{J}$, $\forall i \in J$, $x|_i$ is a fixpoint of $O_i^{x|_{\prec i}}$. By Proposition 3, such a component $O_i^{x|_{\prec i}}$ is equal to $(O_J^{x|_{I \setminus J}})_i^{x|_{\{j \in J | j \prec i\}}}$. By Proposition 2 and Theorem 1, $\forall J \in \mathcal{J}$, $\forall i \in J$, $x|_i$ is a fixpoint $(O_J^{x|_{I \setminus J}})_i^{x|_{\{j \in J | j \prec i\}}}$ iff $\forall J \in \mathcal{J}$, $x|_J$ is a fixpoint of $O_J^{x|_{I \setminus J}}$.

4 Modularity Results for ID-Logic

Now, we apply the algebraic results presented in Section 3 to ID-logic. We fix a vocabulary Σ and a pre-interpretation H for Σ. Also, we restrict our attention to H-interpretations, which can therefore be viewed as sets of domain atoms.

The basic notion needed to split an ID-logic theory, is that of a *dependence relation* between domain atoms. Roughly speaking, such a relation is supposed to express which domain atoms $Q(\mathbf{c})$ can influence whether an operator $\mathcal{T}^O_\Delta$ will derive a certain $P(\mathbf{a})$ in a pair (X, Y). We require that dependence relations are well-founded.

Definition 7. *A well-founded pre-order $\leqslant$ on domain atoms is called a* dependence relation. *We denote by $\mathcal{E}^{\leqslant}$ the set of all equivalence classes $\overline{P(\mathbf{a})} = \{Q(\mathbf{c}) \mid P(\mathbf{a}) \leqslant Q(\mathbf{c}) \text{ and } Q(\mathbf{c}) \leqslant P(\mathbf{a})\}$, together with the well-founded order $\preccurlyeq$, defined as $\overline{P(\mathbf{a})} \preccurlyeq \overline{Q(\mathbf{c})}$ iff $P(\mathbf{a}) \leqslant Q(\mathbf{c})$.*

Such a dependence relation now gives us a product lattice in which to study stratifiability of the operators $\mathcal{T}^O_\Delta$. Recall that we can only apply the algebraic

splitting results, if $\mathcal{T}^O_\Delta$ can be seen as operating on the square of some product lattice $\otimes_{i\in I} L_i$. It turns out that the product of the powersets of all equivalence classes E in $\mathcal{E}^\leqslant$ can give us such a lattice. We denote by $\mathcal{S}^\leqslant$ the lattice $\bigotimes_{E\in\mathcal{E}^\leqslant} 2^E$. Now, $\mathcal{S}^\leqslant$ is isomorphic to the powerset of all domain atoms, which is in turn isomorpic to the set of all H-interpretations. An operator $\mathcal{T}^O_\Delta$ can therefore be seen as operating on the square of the set $\mathcal{S}^\leqslant_O$ of all elements of $\mathcal{S}^\leqslant$ which extend O (or, more precisely, whose image under the appropriate isomorphism extends O).

When dealing with the definition Δ_{even} from Example 1, we will consider the obvious pre-interpretation $H_\mathbb{N}$ with domain $\mathbb{N}$. The set of domain atoms then consists of $\{Even(n) \mid n \in \mathbb{N}\} \cup \{Odd(n) \mid n \in \mathbb{N}\}$. We will use the dependence relation $\leqslant$ consisting of: $Odd(n) \leqslant Even(n+1)$ and $Even(n) \leqslant Odd(n+1)$, for all $n \in \mathbb{N}$. The fact that $\leqslant$ is well-founded follows from the fact that $\mathbb{N}$ is well-founded. The set $\mathcal{E}^\leqslant$ consists of the equivalence classes $\{\overline{Even(n)} \mid n \in \mathbb{N}\} \cup \{\overline{Odd(n)} \mid n \in \mathbb{N}\}$, which are all singletons, i.e., for all $n \in \mathbb{N}$, $\overline{Even(n)} = \{Even(n)\}$ and $\overline{Odd(n)} = \{Odd(n)\}$. The relation $\preccurlyeq$ consists of the pairs $\overline{Even(n)} \preccurlyeq \overline{Odd(n+1)}$ and $\overline{Odd(n)} \preccurlyeq \overline{Even(n+1)}$ with $n \in \mathbb{N}$.

Definition 8. *A dependence relation* $\leqslant$ stratifies *a definition* Δ *given an* H*-interpretation* O *of* Σ^o_Δ *iff the operator* $\mathcal{T}^O_\Delta$ *is a stratifiable approximation on the product lattice* $\mathcal{S}^\leqslant_O$*.*

In [DT04b], a dependence relation that stratifies a definition, is called a *reduction relation.* In case of our example, the dependence relation $\leqslant$ defined above stratifies Δ_{even}. Now, the results presented in Section 3 can be used to show the equivalence of a definition Δ and certain partitions of Δ.

Definition 9. *Let* Δ *be a definition and let* $\leqslant$ *be a dependence relation. A partition* $\{\Delta_1, \ldots, \Delta_n\}$ *of* Δ *is a* $\leqslant$-partition *iff, for each* $1 \leq j \leq n$*, if* Δ_j *contains a rule defining a predicate* P*, then* Δ_j *also contains all rules defining a predicate* Q*, for which there exist tuples* $\mathbf{a}, \mathbf{c}$ *of domain elements, such that* $Q(\mathbf{c}) \in \overline{P(\mathbf{a})}$*.*

In order to show the desired equivalence, we relate the concept of $\leqslant$-partitions to that of recombinations.

Proposition 4. *Let* Δ *be a definition, let* $\leqslant$ *be a dependence relation, and* $\{\Delta_1, \ldots, \Delta_n\}$ *a* $\leqslant$*-partition. Let* O *be an* H*-interpretation of* $\Sigma^o_{\Delta_j}$*, for some* $1 \leq j \leq n$*. Then* $\mathcal{T}^O_{\Delta_j}$ *is equal to the recombination* $(\mathcal{T}^{O_1}_\Delta)^{O_2}_J$*, with* $O_1 = O|_{\Sigma^o_\Delta}$*,* $O_2 = O|_{(\Sigma^o_{\Delta_j} \setminus \Sigma^o_\Delta)}$*, and* $J = \{\overline{P(\mathbf{a})} \mid \Delta_j$ *defines* $P\}$*.*

Proof. Let $\mathcal{T}^O_{\Delta_j}$ and $(\mathcal{T}^{O_1}_\Delta)^{O_2}_J$ be as above. We first note that an H-interpretation X extends O iff it extends $O_1 \sqcup O_2$. It now follows directly from the definitions of the two operators, that $\mathcal{T}^O_{\Delta_j} = (\mathcal{T}^{O_1}_\Delta)^{O_2}_J$ iff for all X, Y extending O, the following two statements are equivalent:

- There exists a rule $\forall \mathbf{x}\ P(\mathbf{t}) \leftarrow \phi$ in Δ_j, for which there exists a $\mathbf{c} \in H^n_D$, such that $(X, Y)[\mathbf{x}/\mathbf{c}] \models \phi$.

– There exists a rule $\forall \mathbf{x}\ P(\mathbf{t}) \leftarrow \phi$ in Δ, for which there exists a $\mathbf{c} \in H_D^n$, such that $(X,Y)[\mathbf{x}/\mathbf{c}] \models \phi$.

Because, for each $P \in \mathcal{P}^d_{\Delta_j}$, Δ_j contains precisely all rules from Δ defining P, this is the case.

As a direct consequence of this proposition and Theorem 2, we now have the following equivalence between a definition and its $\leqslant$-partitions:

Theorem 3. *Let Δ be a definition, $\leqslant$ a dependence relation, and $\{\Delta_1, \dots, \Delta_n\}$ a $\leqslant$-partition. Let O be a Σ-structure, such that $\leqslant$ stratifies Δ given O. Then for each Σ-structure S, such that $S|_{\Sigma^o_\Delta} = O|_{\Sigma^o_\Delta}$:*

$$S \models \Delta \textit{ iff } S \models \Delta_1 \wedge \cdots \wedge \Delta_n.$$

[DT04b] contains a theorem which corresponds to the restriction of this theorem to those cases where each Δ_j is total given O. Our result is strictly more general.

We can now use this result to split the example Δ_{even}. Recall that above we already defined a dependence relation $\leqslant$ which stratifies Δ_{even}. A corresponding $\leqslant$-partition of Δ_{even} is:

$$\Delta_1 = \left\{ \begin{array}{c} Even(0). \\ \forall x\ Even(s(x)) \leftarrow Odd(x). \end{array} \right\}$$
$$\Delta_2 = \left\{ \forall x\ Odd(s(x)) \leftarrow Even(x). \right\}$$

Therefore, for every H-interpretation S, $S \models \Delta_{even}$ iff $S \models \Delta_1 \wedge \Delta_2$.

We now characterize the components of the operators $\mathcal{T}^O_\Delta$ in more detail. Recall that a stratifiable operator $\mathcal{T}^O_\Delta$ has a component $(\mathcal{T}^O_\Delta)^{(U,V)}_E$ for each level $E \in \mathcal{E}^{\leqslant}$ and (U,V) in $(\mathcal{S}^{\leqslant}_O|_{\prec E})^2$. Our goal is now to find a way of deriving some new definition $\Delta^{(U,V)}_E$ from Δ, which characterizes such a component, i.e., such that $(\mathcal{T}^O_\Delta)^{(U,V)}_E = (U_{\Delta^{(U,V)}_E}, U_{\Delta^{(V,U)}_E})$.

Intuitively, there are two main steps in constructing a component-definition $\Delta^{(U,V)}_E$. First, we need to ground Δ w.r.t. to the set of domain atoms E. To do this, we need to assume domain closure, i.e., that for each $a \in H_D$, there exists some term t of Σ, such that $t^H = a$. Such a term is denoted $\hat{a}$; for a tuple $\mathbf{a} = (a_1, \dots, a_n) \in H_D^n$, we denote $(\hat{a}_1, \dots, \hat{a}_n)$ by $\hat{\mathbf{a}}$. Roughly speaking, in the grounding step, a rule r should be replaced by all rules that can be obtained by replacing the universally quantified variables $\mathbf{x}$ of r by some $\hat{\mathbf{a}}$, such that the head of this new rule corresponds to a domain atom in E. Additionally, existential quantifiers also need to be eliminated; this can be done by replacing such a quantifier by a disjunction over all domain elements.

In the following definition, the notation $\phi[\mathbf{x}/\mathbf{y}]$ is used to denote the result of substituting in ϕ every free occurence of a symbol $x \in \mathbf{x}$ by the corresponding symbol $y \in \mathbf{y}$.

Definition 10. *Let Δ be a definition, $E \in \mathcal{E}^{\leqslant}$. For a rule $(\forall \mathbf{x}\ A \leftarrow \phi) \in \Delta$ and domain tuple $\mathbf{a}$, the rule $r^{\mathbf{a}}$ is the rule $A' \leftarrow \phi'$, with $A' = A[\mathbf{x}/\hat{\mathbf{a}}]$ and $\phi' = \gamma(\phi[\mathbf{x}/\hat{\mathbf{a}}])$, with γ defined as:*

- *for each atom A, $\gamma(A) = A$;*
- *$\gamma(\phi_1 \vee \phi_2) = \gamma(\phi_1) \vee \gamma(\phi_2)$ and $\gamma(\neg\phi) = \neg\gamma(\phi)$;*
- *$\gamma(\exists x\ \phi) = \bigvee_{a \in H_D} \gamma(\phi[x/\hat{a}])$;*

The grounding $\lfloor r \rfloor_E$ *of a rule $r = (\forall \mathbf{x}\ P(\mathbf{t}) \leftarrow \phi) \in \Delta$, is the set of rules $r^{\mathbf{a}}$, with $\mathbf{a}$ a domain tuple, such that $P(\mathbf{t}^{H[\mathbf{x}/\mathbf{a}]}) \in E$. The* grounding $\lfloor \Delta \rfloor_E$ *of Δ is $\bigcup_{r \in \Delta} \lfloor r \rfloor_E$.*

In a second step, we now replace ground atoms $P(\mathbf{t})$ for which $\overline{P(\mathbf{t}^H)} \prec E$, by their truth-value according to (U, V); atoms such that $\overline{P(\mathbf{t}^H)} \in E$ are left as they are. We make the small technical assumption that two predicate symbols T and F exist, such that T holds and F does not.

Definition 11. *Let Δ be a definition, $E \in \mathcal{E}^{\leqslant}$, and $(U, V) \in (\mathcal{S}_O^{\leqslant}|_{\prec E})^2$. For each rule $r = (A \leftarrow \phi) \in \lfloor \Delta \rfloor_E$, we define $r^{(U,V)}$ as the rule $A \leftarrow \delta^{(U,V)}(\phi)$, with $\delta^{(U,V)}$ inductively defined as:*

- *for each atom $A = P(\mathbf{t})$, such that $P(\mathbf{t}^H) \notin E$:*
  $\delta^{(U,V)}(A)$ is T if $(U, V) \models A$ and F otherwise;
- *for each other atom A, $\delta^{(U,V)}(A) = A$;*
- *$\delta^{(U,V)}(\phi_1 \vee \phi_2) = \delta^{(U,V)}(\phi_1) \vee \delta^{(U,V)}(\phi_2)$;*
- *$\delta^{(U,V)}(\neg\phi) = \neg\delta^{(V,U)}(\phi)$.*

We define $\Delta_E^{(U,V)}$ as $\{r^{(U,V)} \mid r \in \lfloor \Delta \rfloor_E\}$.

The proof of the following theorem is omitted, as it follows easily from the various definitions.

Theorem 4. *Let Δ be a definition, $E \in \mathcal{E}^{\leqslant}$, $U, V \in \mathcal{S}_O^{\leqslant}|_{\prec E}$, and O an H-interpretation of Σ_Δ^o. Then $(U_\Delta^O)_E^{(U,V)} = U^O_{\Delta_E^{(U,V)}}$ and $(U_\Delta^O)_E^{(V,U)} = U^O_{\Delta_E^{(V,U)}}$.*

Let us look again at definition Δ_{even} from Example 1, with the obvious pre-interpretation $H_{\mathbb{N}}$. If $E = \{Even(n+1)\}$ for some $n \in \mathbb{N}$, then for all $U, V \in \mathcal{S}^{\leqslant}|_{\prec E}$ the component $(\mathcal{T}_{\Delta_{even}})_E^{(U,V)}$ is the constant function $\{Even(n+1)\}$ if $n \in Odd^U$ and the constant function $\{\}$ otherwise. Similarly, for every level $E = \{Odd(n+1)\}$, $(\mathcal{T}_{\Delta_{even}})_E^{(U,V)}$ is the constant function $\{Odd(n+1)\}$ if $n \in Even^U$ and the constant function $\{\}$ otherwise. The component $(\mathcal{T}_{\Delta_{even}})_{\{Even(0)\}}$ is the constant function $\{Even(0)\}$, while the component $(\mathcal{T}_{\Delta_{even}})_{\{Odd(0)\}}$ is the constant function $\{\}$. From this, it follows that there exists a unique model of Δ_{even} extending $H_{\mathbb{N}}$, namely that which interprets $Even$ by $\{n \in \mathbb{N} \mid n \text{ is even}\}$ and Odd by $\{n \in \mathbb{N} \mid n \text{ is odd}\}$.

While space restrictions prevent us from discussing this here, this characterization of the components of a stratifiable operator promises to be useful for the study of the relation between ID-logic and known classes of mathematical inductive definitions. For instance, we suspect that the class of *well-founded inductions* coincides precisely with the class of ID-logic definitions whose $\mathcal{T}_\Delta^O$-operators can be split into constant components, as witnessed by the above example.

5 Conclusions and Related Work

Our work extends that from [VGD04b, VGD04a] about algebraic modularity results. Firstly, we have extended these results to also allow operators to be split into recombinations, rather than components. Secondly, our work is the first to apply these results outside a propositional context.

Our work also extends previous work on modularity properties for ID-logic [DT04b], by generalizing existing results in Theorem 3 and by the additional Theorem 4. It is interesting to note that, although in the context of ID-logic we are only interested in the well-founded fixpoints of the operators associated with definitions, our results also suffice to show a similar correspondence between their Kripke-Kleene and stable fixpoints. Indeed, this follows directly from the generality of the algebraic splitting theorem (Theorem 1). As such, our work actually also generalizes the results from [VGD04b], which in turn generalized part of the splitting theorem for the stable model semantics from [LT94].

The work presented here demonstrates that approximation theory and algebraic modularity results can be used to elegantly and easily derive useful results, even in a complex setting. In our opinion, it therefore offers quite a convincing testimony to the power of this approach.

References

[Den01] M. Denecker. Logic programming revisited: logic programs as inductive definitions. In *ACM Transactions on Computational Logic*, 2(4):623-654, 2001.

[DMT00] M. Denecker, V. Marek, and M. Truszczynski. Approximating operators, stable operators, well-founded fixpoints and applications in non-monotonic reasoning. In *Logic-based Artificial Intelligence*, The Kluwer International Series in Engineering and Computer Science, pages 127–144, 2000.

[DMT03] M. Denecker, V. Marek, and M. Truszczynski. Uniform semantic treatment of default and autoepistemic logics. *Artificial Intelligence*, 143(1):79–122, 2003.

[DT04a] M. Denecker and E. Ternovska. Inductive situation calculus. In *Principles of Knowledge Representation and Reasoning: Proceedings of the Ninth International Conference (KR2004)*, pages 545–553. AAAI Press, 2004.

[DT04b] M. Denecker and E. Ternovska. A logic of non-monotone inductive definitions and its modularity properties. In *7th International Conference on Logic Programming and Nonmonotonic Reasoning*, 2004.

[GP92] M. Gelfond and H. Przymusinska. On consistency and completeness of autoepistemic theories. *Fundamenta Informaticae*, 16(1):59–92, 1992.

[LT94] V. Lifschitz and H. Turner. Splitting a logic program. In *Proceedings of the 11th International Conference on Logic Programming*, pages 23–37, 1994.

[VGD04a] J. Vennekens, D. Gilis, and M. Denecker. Splitting an operator: An algebraic modularity result and its application to auto-epistemic logic. In *Proceedings of International Workshop on Non-Monotonic Reasoning*, 2004.

[VGD04b] J. Vennekens, D. Gilis, and M. Denecker. Splitting an operator: An algebraic modularity result and its applications to logic programming. In *Logic Programming, 20th International Conference, ICLP 2004, Proceedings*, volume 3132 of *Lecture Notes in Computer Science*, pages 195–209. Springer, 2004.

Default Reasoning with Preference Within Only Knowing Logic

Iselin Engan[1], Tore Langholm[1], Espen H. Lian[2], and Arild Waaler[2,3]

[1] Dept. of Linguistics, University of Oslo, Norway
[2] Dept. of Informatics, University of Oslo, Norway
[3] Finnmark College, Norway

Abstract. The main construction in this paper is an encoding of default logic into an "only knowing" logic with degrees of confidence. By imposing simple and natural constraints on the encoding we show that the "only knowing" logic can accommodate ordered default theories and that the constrained encoding implements a prescriptive interpretation of preference between defaults. An advantage of the encoding is that it provides a transparent formal rendition of such a semantics. A feature of the construction is that the generation of extensions can be carried out within the "only knowing" logic, using object level concepts alone.

1 Introduction

Although default logic [16] is most widely studied among the consistency-based approaches to non-monotonic reasoning, a number of arguments have emerged in favour of an autoepistemic approach [15]. In autoepistemic logic, defaults are provided with a clear model-theoretic semantics, since they are represented by modal belief formulae, as opposed to rules. A further clarification of the semantics of autoepistemic logic was provided by Levesque's introduction of "only knowing" logic [12], by which he achieved a clearer separation of the object- and meta-level parts of the modal language. Thus, in Levesque's logic a notion of e.g. consistency of defaults receives a clear interpretation. Another advantage of an autoepistemic approach over standard default logic is that the underlying language can be extended to represent multi-agent reasoning.

These arguments carry over to the debate on how to represent preference in non-monotonic reasoning. The notion of preference has been added to the framework of non-monotonic reasoning as one way of avoiding unintended ambiguities, known as the multiple extension problem. A preference order intuitively captures the idea that some defaults lead to more plausible conclusions than others. Two varieties of preference have been identified [5]: *prescriptive,* which "prescribes" the order in which to apply defaults, and *descriptive,* which is defined independently of application order. E.g. preferring the default $p : q \,/\, q$ to the default $\top : p \,/\, p$, without further world beliefs, prescriptively yields no extension, because applying the more preferred default requires us to apply the less preferred default first. Descriptively, however, an extension including p and q is

C. Baral et al. (Eds.): LPNMR 2005, LNAI 3662, pp. 304–316, 2005.
© Springer-Verlag Berlin Heidelberg 2005

accepted. This paper adopts a prescriptive interpretation, because descriptive approaches seem to require extra meta-level machinery [6].

In recent years, a number of approaches to handling preference have been proposed in terms of default logic [1,2,5,3]. In contrast, only a few approaches interpret preference in relation to autoepistemic logic [10,17]. Rintanen's *prioritized autoepistemic logic* interprets preference descriptively, thus imposing a higher complexity on the system, while *hierarchic autoepistemic logic* (HAEL), introduced by Konolige, is prescriptive, and avoids this disadvantage. HAEL is founded on the idea of a hierarchy of beliefs. It gives a natural interpretation of prescriptive preference in autoepistemic logic because it captures the idea that some beliefs take precedence over others. In HAEL, the multiple extension problem is avoided in the sense that each theory gives rise to a unique extension.

However, Konolige's system fails to satisfy a number of evaluative principles that have recently emerged in the default literature [6] and which, e.g., the prescriptive semantics of preference satisfies. In light of these principles, HAEL is not a satisfactory approach to preference. One shortcoming is that a theory may in HAEL give rise to extensions that are not extensions of the underlying classical default theory. The system also violates the weak Principle I proposed by Brewka and Eiter [3], cf. Sect. 2.2 below. An even more unfortunate aspect of HAEL is its limited expressive power, a result of disallowing nested modalities. Thus, the notion of an agent reasoning about his own beliefs cannot be represented.

Recently Levesque's system has been generalized to the logic Æ of "only knowing" with degrees of confidence [13] which does not suffer from these shortcomings. The added expressive power is illustrated in [13] by a simple representation in Æ of supernormal defaults equipped with a preference relation. The computation of extensions is carried out at the object level by means of equivalence-preserving rewriting steps to yield a formula of a particularly simple form, from which the extensions of the theory are straightforwardly identified.

In this paper we show that there is a translation of any ordered default theory with prescriptive semantics into Æ. A central idea underlying our work is that modalities can be used to control deduction. We here control deduction by assigning different modalities to different steps in the generation of an extension, and impose logical relationships between the modalities which reflect the intended interpretation of the modalities. This allows us to give a precise formal rendition of the semantics of prescriptive preference, characterized by the property of order-preservation from the approach of Delgrande and Schaub [5]. In contrast to the their approach, our formalization has the advantage that all aspects of the default theory are formulated entirely at the object level. Seeing that a substantial part of the reasoning used to *generate* extensions is captured within the logic itself, a higher degree of formalization is obtained. Another contribution of this work is that the semantics of prescriptive ordered defaults is expressed in a simpler and more transparent way than the technical encoding provided in the work of Delgrande and Schaub.

The exact relation between autoepistemic logic and default logic is non-trivial and has been addressed in a number of studies [4,9,11,14,7]. From the results

of Gottlob [8] we know that an autoepistemic representation of default logic cannot be both faithful and modular, hence a solution to this task is far from straightforward. This picture changes significantly when we instead represent ordered default theories in autoepistemic logic. Since our approach treats classical default theories as ordered theories with empty preference order, it follows that there is a simple translation of classical default logic into an autoepistemic system with language constructs representing different degrees of confidence.

2 Preliminaries

We will assume a propositional language over a finite set Φ of propositional variables, the Boolean constants $\bot$ and $\top$, and the usual connectives. Formulae in this language are called *purely Boolean.* If S is a set of purely Boolean formulae, $\mathrm{Th}(S)$ denotes the closure of S under classical consequence.

2.1 Default Logic

A *classical default theory* is a pair (W, D) consisting of a set W of purely Boolean formulae and a finite set D of default rules or *defaults.* In a default $\delta = \alpha : \beta \,/\, \gamma$ the *prerequisite* α is denoted $\mathrm{pre}(\delta)$, the *justification* β is denoted $\mathrm{just}(\delta)$, while $\mathrm{cons}(\delta)$ is the *consequent* γ. Default theories induce zero or more extensions; for any (W, D) this notion is defined from the function $\Gamma_{(W,D)}$ which maps any set S of purely Boolean formulae to the smallest set E for which (1) $W \subseteq E$, (2) $E = \mathrm{Th}(E)$ and (3) $\gamma \in E$ for each $\alpha : \beta \,/\, \gamma$ in D such that $\alpha \in E$ and $\neg\beta \notin S$. Following Reiter [16] we say that the *extensions* of a classical theory $\Delta = (W, D)$ are the fixed-points of Γ_Δ, i.e. those S for which $\Gamma_\Delta(S) = S$.

Extensions can also be characterized in a more constructive way along the following lines. Let (W, D) be a default theory; a *generating sequence* is a sequence u of defaults in D, not containing repetitions, such that for any $\delta \in D$

GS1 if $s\delta \preceq u$ then $W \cup \mathrm{cons}(s) \vdash \mathrm{pre}(\delta)$ and $W \cup \mathrm{cons}(u) \nvdash \neg\mathrm{just}(\delta)$, and
GS2 δ occurs in u if $W \cup \mathrm{cons}(u) \vdash \mathrm{pre}(\delta)$ and $W \cup \mathrm{cons}(u) \nvdash \neg\mathrm{just}(\delta)$.

Here, $\mathrm{cons}(u)$ is the set of consequents of defaults in u, while $s \preceq u$ denotes that s is an initial segment of u. By $s\delta$ we mean the sequence s with δ added to the right. The next lemma is a straightforward consequence of Theorem 2.1 of [16].

Lemma 1. *The extensions of* (W, D) *are the sets* $Th(W \cup cons(u))$ *for generating sequences* u.

For sequences s of defaults we define formulae ψ_s by recursion as follows.

$$\psi_\varepsilon = \bigwedge W$$
$$\psi_{s\delta} = \begin{cases} \psi_s \bigwedge \mathrm{cons}(\delta) & \text{if } \psi_s \vdash \mathrm{pre}(\delta) \text{ and } \psi_s \nvdash \neg\mathrm{just}(\delta) \\ \psi_s & \text{otherwise} \end{cases}$$

Hence $Th(W \cup cons(u)) = Th(\psi_u)$ for any generating sequence u (or any initial segment of one) while $Th(\psi_u)$ in general (i.e., for arbitrary sequences of defaults) is only a subset of $Th(W \cup cons(u))$. An enumeration[1] t of D is *felicitous* if

$$\psi_t \vdash \text{pre}(\delta) \text{ and } \psi_s \nvdash \neg\text{just}(\delta) \text{ together imply } \psi_s \vdash \text{pre}(\delta) \text{ and } \psi_t \nvdash \neg\text{just}(\delta)$$

whenever $s\delta \preceq t$. This notion can be used to characterize extensions in a way similar to generating sequences, as seen in the next lemma, which follows as a special case of Lemma 3.

Lemma 2. *The extensions of* (W, D) *are the sets* $Th(\psi_t)$ *for felicitous enumerations* t *of* D.

2.2 Ordered Default Theories

An *ordered default theory* is a structure $(W, D, <)$ such that (W, D) is a classical default theory and $<$ is a strict partial order on D. $\delta_1 < \delta_2$ intuitively expresses that δ_1 is *preferred* to δ_2. The partiality of the preference relation is important as there are cases where the defaults are incomparable.

Following Delgrande and Schaub [5], we define an *extension* of an ordered theory[2] $(W, D, <)$ as an order-preserving extension of the underlying classical default theory, where an extension E of (W, D) is defined to be *order-preserving* if there is a generating sequence u such that $E = \text{Th}(\psi_u)$, which satisfies the following conditions[3]

for each initial segment $r\delta \preceq u$ and each default $\delta' \in D$:

O1 if $\delta < \delta'$, then δ' is not in r,
O2 if $\delta' < \delta$ and $\psi_u \vdash \text{pre}(\delta')$ and $\psi_r \nvdash \neg\text{just}(\delta')$, then δ' is in r.

O1 says that a less preferred default is never to be applied before a more preferred one, and O2 says (contrapositively) that if, at any point (r), the justification of a more preferred default (δ') that has not yet been applied is still possible, and the prerequisite of this default is eventually believed, then a less preferred default (δ) shall not be used in the next step ($r\delta \not\preceq u$). The next result says that this notion of extension is fully determined from its behaviour on strict linear orderings.

Lemma 3. *E is an extension of the ordered theory* $(W, D, <)$ *iff E is an extension of* $(W, D, <')$ *for some strict total ordering* $<'$ *on* D *containing* $<$.

Proof. As $<$ only occurs negatively in O1 and O2, the "if" direction is trivial. The "only if" direction assumes an arbitrary generating sequence u satisfying O1

[1] An enumeration of D is a sequence without repetitions, containing all and only the elements of D.

[2] The authors interpret the preference relation ¡ in the opposite direction from what is done here, in accordance with the convention of answer set programming. We, on the other hand, follow the convention of [3].

[3] It is worth noting that in the presence of O1, O2 is equivalent to a version with the weaker conclusion that δ' is in u.

and O2 with respect to $<$, and proceeds by construction of a strict linear order $<'$ extending $<$, with respect to which u still satisfies O1 and O2. To ensure this, $<'$ is chosen so that it agrees with the ordering of elements in u and also, whenever $<$ allows, gives precedence to u-elements over non-u-elements.[4] □

An enumeration of D satisfying O1 is said to be a *topological sorting* of $(D, <)$. The encoding introduced in Sect. 3 relies on this notion; the soundness of the approach rests on the following observation.

Lemma 4. *The extensions of* $(W, D, <)$ *are the sets* $Th(\psi_t)$ *for felicitous enumerations* t *of* D *which are also topological sortings of* $(D, <)$.

Proof. The latter, alternative, notion of extension clearly satisfies the property of Lemma 3, hence it suffices to prove that the two notions coincide on any $(W, D, <)$ where $<$ is a strict linear ordering on D.

For such $(D, <)$ we can suppose that $D = \{\delta_1, \cdots, \delta_n\}$, where $\delta_i < \delta_j$ iff $i < j$. Now in the presence of O1 a generating sequence u is fully determined from the corresponding subset U of D, hence GS1 and GS2 can be rephrased to 1 and 2 below, while the additional content of O2 is represented by 3. [5] We suppose that $\delta_i = \alpha_i : \beta_i/\gamma_i$ for all i.

1. $\delta_j \in U \;\Rightarrow\; (W \cup \{\gamma_i \mid \delta_i \in U, i < j\} \vdash \alpha_j \;\&\; W \cup \{\gamma_i \mid \delta_i \in U\} \not\vdash \neg\beta_j)$,
2. $\delta_j \in U \;\Leftarrow\; (W \cup \{\gamma_i \mid \delta_i \in U\} \vdash \alpha_j \;\&\; W \cup \{\gamma_i \mid \delta_i \in U\} \not\vdash \neg\beta_j)$,
3. $\delta_j \in U \;\Leftarrow\; (W \cup \{\gamma_i \mid \delta_i \in U\} \vdash \alpha_j \;\&\; W \cup \{\gamma_i \mid \delta_i \in U, i < j\} \not\vdash \neg\beta_j)$.

Now 3 of course implies 2, and it is easily checked that 1 and 3 in combination are equivalent to the implication from the right-hand side of 3 to the right-hand side of 1, in combination with the equivalence

$$\delta_j \in U \;\Leftrightarrow\; (W \cup \{\gamma_i \mid \delta_i \in U, i < j\} \vdash \alpha_j \;\&\; W \cup \{\gamma_i \mid \delta_i \in U, i < j\} \not\vdash \neg\beta_j).$$

Now recall that we are existentially quantifying over $U \subseteq D$. As $\delta_1 \cdots \delta_n$ is the only topological sorting of D, we see that the two latter conditions (with U existentially quantified) precisely express that $\delta_1 \cdots \delta_n$ is felicitous. □

[4] If $u = \delta_1 \cdots \delta_n$ then for $i = 0, \ldots, n$ let A_i be $\{\delta \notin u \mid \delta < \delta_1 \vee \cdots \vee \delta < \delta_i\}$ (hence $A_0 = \emptyset$) while $A_{n+1} = \{\delta \mid \delta \notin u\}$, and for $i = 1, \ldots, n+1$ let B_i be $A_i \setminus A_{i-1}$. Then let $<'$ be any strict, linear order containing $<$, such that all elements of B_1 precede δ_1, which in turn precedes all elements of B_2, which all precede δ_2, etc. Inside each B_i, the defaults can be ordered in any way that agrees with $<$.

[5] The direct rendering of O2 (actually, the equivalent (modulo O1) version noted in footnote 3) is rather that if $k > j$ for any $\delta_k \in U$ then $\delta_i \in U$ if $W \cup \{\gamma_i \mid \delta_i \in U\} \vdash \alpha_j$ and $W \cup \{\gamma_i \mid \delta_i \in U, i < k\} \not\vdash \neg\beta_j$. If there are several such k, the implication obtained from the smallest one yields the corresponding implications for all larger, hence provided there *is* such a $k > j$ for which $\delta_k \in U$, O2 is equivalent to the condition 3' below.

$$\delta_j \in U \;\Leftarrow\; (W \cup \{\gamma_i \mid \delta_i \in U\} \vdash \alpha_j \;\&\; W \cup \{\gamma_i \mid \delta_i \in U, i \leq j\} \not\vdash \neg\beta_j)$$

If there is *no* such k, 3' is equivalent to 2, hence in either case O2 is, in the presence of 2, equivalent to 3'. Finally, the falsifying conditions of 3 and 3' are clearly identical.

From properties noted below and the fact that any enumeration is a topological sorting for empty $<$, we see that Lemma 2 is a special case of Lemma 3.

In the literature, approaches to adding preference to a non-monotonic logic have proved to yield very different results. Therefore, certain principles and properties that serve to evaluate an ordered logic have emerged. The following three principles follow easily from Lemma 2 and Lemma 4.

- *Classical Subset*: Every extension of an ordered default theory $(W, D, <)$ is an extension of the classical default theory (W, D).
- *Empty order*: Every extension of the classical default theory (W, D) is an extension of the ordered default theory $(W, D, \emptyset)$.
- *Inapplicable defaults*: Any extension E of an ordered default theory $(W, D, <)$ is also an extension of the ordered default theory $(W, D \cup \{\delta\}, <')$ if δ is a default such that $\text{pre}(\delta) \notin E$ and $<'$ is a conservative extension of $<$.

The first two principles were identified in [6]. The third was identified in [3] and is there referred to as Principle II. The corresponding Principle I can be formulated in the following way. Let E_1 and E_2 be two extensions of a classical default theory (W, D) and let u_1 and u_2 be generating sequences s.t. $E_1 = \text{Th}(W \cup \text{cons}(u_1))$ and $E_2 = \text{Th}(W \cup \text{cons}(u_2))$. Assume that, neglecting their intrinsic order, u_1 and u_2 only differ by the defaults δ_1 and δ_2. If $\delta_1 < \delta_2$, then E_2 is not an extension of $(W, D, <)$.

This principle is easily seen to hold for our definition of extension: if E_1, E_2, u_1, u_2, δ_1, δ_2 are as above and r_1, r_2 are the sequences such that $r_1\delta_1 \preceq u_1$ and $r_2\delta_2 \preceq u_2$ then, as u_1 is a generating sequence, we have $W \cup \text{cons}(r_1) \vdash \text{pre}(\delta_1)$ and $W \cup \text{cons}(u_1) \not\vdash \neg\text{just}(\delta_1)$. Now since every default in r_1 is in u_2 and every default in r_2 is in u_1, we also infer $W \cup \text{cons}(u_2) \vdash \text{pre}(\delta_1)$ and $W \cup \text{cons}(r_2) \not\vdash \neg\text{just}(\delta_1)$. But then u_2 is not order-preserving, since $\delta_1 < \delta_2$ and $\delta_1 \notin u_2$.

2.3 The Logic $\text{Æ}_\top$

The system $\text{Æ}_\top$ belongs to the family of "only knowing" logics. It generalizes the pioneering system of Levesque [12] to a language which allows us to represent various degrees of confidence for a doxastic subject. $\text{Æ}_\top$ is a special instance of the system Æ_ρ introduced in [13] and further analysed and motivated in [20]. An interesting proof-theoretical property of $\text{Æ}_\top$ is that it has a sequent calculus formulation which admits constructive cut-elimination and hence cut-free proofs; this is shown in [18] for a generalization to a multi-agent language in which the beliefs of each subject are represented relative to different degrees of confidence.

This section contains a review of the main concepts of $\text{Æ}_\top$. The object language extends the language of purely Boolean formula by the addition of modal operators: $\Box$ (necessity) and modalities B_k (belief) and C_k (co-belief) for each k in a finite index set I. I comes along with a strict partial order $\lhd$. $b_k\varphi$ is defined as $\neg B_k \neg\varphi$ and denotes that φ is compatible with belief at k. A formula φ is *completely modalized* if every occurrence of a propositional letter is within the scope of a modal operator.

The intuitions behind the syntactical operators are discussed at length in [20]. In short, $\Box$ is intended to express personal necessities. The indices in I are intended to represent various degrees of *confidence* or *conviction*. $B_k\varphi$ expresses that φ is believed with degree of confidence k; $C_k\varphi$ expresses that φ is co-believed with degree of confidence k. The belief and co-belief operators are complementary. $C_k\varphi$ expresses a notion of *caution*, and can generally be read as expressing that *at most* $\neg\varphi$ is believed with degree of confidence k; or, what amounts to the same, that $\neg\varphi$ is at least as strong as everything that is believed at k.

The "all I know at k" expression $O_k\varphi$ is central; it abbreviates $B_k\varphi \wedge C_k\neg\varphi$, meaning that *precisely* φ is believed with degree of confidence k. A formula of the form $\bigwedge_{k\in I} O_k\varphi_k$ is called an O_I-*block*. If each φ_k is purely Boolean, the O_I-block is said to be *prime*.

An Æ$_\top$-*model* M is a quadruple (U, U^+, U^-, V). The universe U is a non-empty set of *points*; U^+ and U^- are functions which assign a subset of U to each index in I. $U^+(k)$ is denoted U_k^+; U_k^- denotes $U^-(k)$. V is a valuation function which assigns a subset of U to each propositional letter in the language. For each $k \in I$, we require that $U_k^+ \cup U_k^- = U$, expressing that while belief states may vary between degrees of confidence, the universe does not. We also require that greater confidence is never accompanied by stronger belief: $U_k^+ \subseteq U_i^+$ and $U_i^- \subseteq U_k^-$ for each $i \lhd k$. Finally we require that U is *Boolean saturated*, i.e., that the following condition holds for every subset P of the propositional letters: there is a point $x \in U$ such that for each propositional letter p, $x \in V(p)$ iff $p \in P$. Informally, the points in U span the set of all propositional valuations. A satisfaction relation can be defined for each point x:

$$
\begin{array}{ll}
M \vDash_x p & \text{iff } x \in V(p), \text{ when } p \text{ is any propositional letter} \\
M \vDash_x \Box\varphi & \text{iff } M \vDash_y \varphi \text{ for each } y \in U \\
M \vDash_x B_k\varphi & \text{iff } M \vDash_y \varphi \text{ for each } y \in U_k^+ \\
M \vDash_x C_k\varphi & \text{iff } M \vDash_y \varphi \text{ for each } y \in U_k^-
\end{array}
$$

and as usual for Boolean connectives. A formula is *satisfied* in a model if it is true at one of its points. If $M \vDash_x \varphi$ for all $x \in U$ we write $M \vDash \varphi$ and say that φ is *true in M*. If φ is true in all models, we also write $\vDash \varphi$. Note that all points in a model agree on the truth value of every completely modalised formula. Hence, for such formulae the notions of satisfiability and truth in a model coincide, and the notation $M \vDash \varphi$ can be used to express either notion. We omit the easy proof of the following useful observation.

Lemma 5. *Let M satisfy $O_k\varphi$ for an index k.*

1. *If M satisfies $O_k\psi$, then $\varphi \equiv \psi$ is true at every point in M.*
2. *If φ and ψ are purely Boolean, M satisfies $B_k\psi$ iff $\varphi \vdash \psi$ and $b_k\psi$ iff $\varphi \nvdash \neg\psi$.*

A model is *bisected* if, for each $k \in I$, $U_k^+ \cap U_k^- = \emptyset$. This is not a model condition. It is however, easy to prove that a model is bisected if it satisfies an O_I-block.

An axiom system for Æ is defined in [13].[6] From here on $\vdash$ denotes the provability relation of $Æ_\top$ (which extends the provability relation of classical logic). The following properties have been established for $Æ_\top$ [13,20]:

- Soundness: if $\vdash \varphi$, then $\models \varphi$.
- Completeness: if $\models \varphi$, then $\vdash \varphi$.
- Decidability and the Finite model property: the logic is determined by the set of finite models.
- The Modal Reduction Theorem: for each O_I-block there is an $m \geq 0$ as well as prime O_I-blocks $\psi_1^I, \ldots, \psi_m^I$ such that $\vdash \varphi^I \equiv (\psi_1^I \vee \cdots \vee \psi_m^I)$.

A prime O_I-block determines the belief state of the agent in a unique and transparent way. It is easy to show that all models of a prime O_I-block are *modally equivalent*: they agree on the truth value of all completely modalised formulae. The models of a non-prime O_I-block are in general not transparent; it requires some work to find them. A non-prime O_I-block hence only implicitly defines the belief state. The Modal Reduction Theorem relates an implicit belief representation to an explicit representation by a provable equivalence. To determine whether $m > 0$ in the statement of the theorem is Σ_2^p-hard.

If there is only one degree of confidence, $Æ_\top$ is equivalent to Levesque's system of only knowing, for which there is a direct correspondence between a stable expansion in autoepistemic logic and a prime formula $O\varphi$. A prime O_I-block is a natural generalization of the notion of stable expansion to an hierarchical collection of expansions.

3 Translating a Default Theory into $Æ_\top$

3.1 The Basic Translation

In this section we define the mapping $[\![\cdot]\!]$ of an ordered default theory $(W, D, <)$ to a formula $[\![W, D, <]\!]$ of $Æ_\top$. The index set I in the signature of $Æ_\top$ is the set of numbers $0, \ldots, |D|$ and the order $\lhd$ on I is the usual strict order on numbers. Let T be the set of topological sortings of $(D, <)$; the formula $[\![W, D, <]\!]$ quantifies over initial segments of elements of T:

$$[\![W, D, <]\!] = \bigvee\nolimits_{t \in T} \left(\bigwedge\nolimits_{s \preceq t} [\![W, D]\!]_s \wedge \mathrm{IC}(t) \right)$$

$$[\![W, D]\!]_s = O_{|s|} \left(\bigwedge W \wedge \bigwedge\nolimits_{r\delta \preceq s} \mathrm{tr}(r\delta) \right)$$

$$\mathrm{IC}(t) = \bigwedge\nolimits_{s\delta \preceq t} ((B_{|t|}\alpha \wedge b_{|s|}\beta) \supset (B_{|s|}\alpha \wedge b_{|t|}\beta))$$

Here we have used the notational convention of identifying δ with $\alpha : \beta \,/\, \gamma$. The *integrity constraint* function IC corresponds exactly to the condition for a felicitous enumeration (cf. the proof of Lemma 7). The function served by tr

[6] $Æ_\top$ is just the system $Æ_\rho$ in [13] with the characteristic formula ρ set to $\top$.

is to map specific defaults. We shall first define one basic mapping and later introduce two alternative translations. The basic mapping is:

$$\mathrm{tr}(r\delta) = (B_{|r|}\alpha \wedge b_{|r|}\beta) \supset \gamma \ .$$

The basic correctness result below relates the translation to formulae ψ_s studied in Sect. 2.1.

Lemma 6. *Let t be any enumeration of D. Then*

$$\vdash \bigwedge\nolimits_{s \preceq t} [\![W, D]\!]_s \equiv \bigwedge\nolimits_{s \preceq t} O_{|s|}\psi_s \ .$$

Proof. We show, by induction on v, the more general result that for any $v \preceq t$,

$$\vdash \bigwedge\nolimits_{s \preceq v} [\![W, D]\!]_s \equiv \bigwedge\nolimits_{s \preceq v} O_{|s|}\psi_s \ .$$

The basis is trivial, as $[\![W, D]\!]_\varepsilon = O_0 \bigwedge W$ and $\psi_\varepsilon = \bigwedge W$. For the induction step, it suffices to show that $M \vDash [\![W, D]\!]_{s\delta} \equiv O_{|s\delta|}\psi_{s\delta}$ for any $\mathrm{Æ}_\top$-model M satisfying both $[\![W, D]\!]_s$ and $O_{|s|}\psi_s$. By Lemma 5(1), every such model M satisfies

$$M \vDash (\bigwedge W \wedge \bigwedge\nolimits_{r\delta' \preceq s} \mathrm{tr}(r\delta')) \equiv \psi_s \ .$$

Thus $M \vDash [\![W, D]\!]_{s\delta} \equiv O_{|s\delta|}(\psi_s \wedge tr(s\delta))$. It only remains to show

$$M \vDash O_{|s\delta|}(\psi_s \wedge ((B_{|s|}\alpha \wedge b_{|s|}\beta) \supset \gamma)) \equiv O_{|s\delta|}\psi_{s\delta} \ .$$

But since $M \vDash O_{|s|}\psi_s$, it follows directly from the definition of $\psi_{s\delta}$ and Lemma 5(2) that $M \vDash B_{|s|}\alpha$ iff $\psi_s \vdash \alpha$, and $M \vDash b_{|s|}\beta$ iff $\psi_s \nvdash \neg\beta$, and we are done. □

If s is an enumeration and $[\![W, D]\!]_s$ is proved equivalent to $O_{|s|}\psi_s$, $\mathrm{Th}\{\psi_s\}$ is a potential extension of the underlying classical theory (W, D). The integrity constraint will, however, reduce to $\top$ precisely when s is felicitous and reduce to $\bot$ otherwise, as illustrated in the examples below.

Example 1 (From [16]). Let $W = \emptyset$, $D = \{\delta\}$ and $\delta = \top : p \,/\, \neg p$. As there is only one default, $<$ is empty. $[\![W, D, <]\!] = [\![W, D]\!]_\varepsilon \wedge [\![W, D]\!]_\delta \wedge \mathrm{IC}(\delta)$, where

$$\begin{aligned}
[\![W, D]\!]_\varepsilon &= O_0\top \\
[\![W, D]\!]_\delta &= O_1(\top \wedge ((B_0\top \wedge b_0 p) \supset \neg p)) \\
\mathrm{IC}(\delta) &= (B_1\top \wedge b_0 p) \supset (B_0\top \wedge b_1 p)
\end{aligned}$$

By Lemma 6, $\vdash [\![W, D, <]\!] \equiv O_0\top \wedge O_1\neg p \wedge (b_0 p \supset b_1 p)$, thus $[\![W, D, <]\!]$ is inconsistent. This illustrates that the integrity constraints serve to exclude potential extensions.

Example 2 (From [3,5]). Let $W = \emptyset$, $D = \{\delta_1, \delta_2, \delta_3\}$, $< = \emptyset$, and

$$\delta_1 = \frac{\top : q}{q}, \quad \delta_2 = \frac{\top : p}{p}, \quad \text{and} \quad \delta_3 = \frac{\top : \neg q}{p} \ .$$

Below are listed every enumeration and a simpler, equivalent form of the translation. Since the order $<$ is empty, the entire translation is the disjunction of the translation for each enumeration in the table.

t	$\bigwedge_{s \preceq t} [\![W, D]\!]_t$	IC(t)	Extension
$\delta_1\delta_2\delta_3$	$O_0\top \wedge O_1 q \wedge O_2(p \wedge q) \wedge O_3(p \wedge q)$	$\wedge\ \top$	Th$\{p, q\}$
$\delta_1\delta_3\delta_2$	$O_0\top \wedge O_1 q \wedge O_2(p \wedge q) \wedge O_3(p \wedge q)$	$\wedge\ \top$	Th$\{p, q\}$
$\delta_2\delta_1\delta_3$	$O_0\top \wedge O_1 p \wedge O_2(p \wedge q) \wedge O_3(p \wedge q)$	$\wedge\ \top$	Th$\{p, q\}$
$\delta_2\delta_3\delta_1$	$O_0\top \wedge O_1 p \wedge O_2 p \wedge O_3(p \wedge q)$	$\wedge\ \bot$	-
$\delta_3\delta_1\delta_2$	$O_0\top \wedge O_1 p \wedge O_2(p \wedge q) \wedge O_3(p \wedge q)$	$\wedge\ \bot$	-
$\delta_3\delta_2\delta_1$	$O_0\top \wedge O_1 p \wedge O_2 p \wedge O_3(p \wedge q)$	$\wedge\ \bot$	-

When t is any of the last three enumerations $\bigwedge_{s \preceq t} [\![W, D]\!]_t \vdash \neg\text{IC}(t)$, as

$$\text{IC}(\delta_2\delta_3\delta_1) \vdash b_1\neg q \supset b_3\neg q \quad \text{and} \quad \text{IC}(\delta_3\delta_1\delta_2) \vee \text{IC}(\delta_3\delta_2\delta_1) \vdash b_0\neg q \supset b_3\neg q.$$

It can be seen that these enumerations are not felicitous while the remaining are and moreover define the extensions in the rightmost column. Note that the entire translation entails $O_3(p \wedge q)$, showing that Th$\{p, q\}$ is the unique extension.

If we change the order relation, the translation selects as disjuncts only the enumerations that are topological sortings of the order. Note that any order where $\delta_3 < \delta_1$ yields no extension, as this constraint rules out the first three enumerations. If we, on the other hand, use the order $\delta_1 < \delta_3$, we get the same extensions as when δ_1 and δ_3 are unrelated. Restricting the order further has no impact on the extensions, as $\delta_1 < \delta_2 < \delta_3$, $\delta_1 < \delta_3 < \delta_2$ and $\delta_2 < \delta_1 < \delta_3$ all yield the extension Th$\{p, q\}$.

Example 3 (Nixon diamond). Let $W = \{q, r\}$, $D = \{\delta_1, \delta_2\}$, and

$$\delta_1 = \frac{q : p}{p} \quad \text{and} \quad \delta_2 = \frac{r : \neg p}{\neg p} \ .$$

t	$\bigwedge_{s \preceq t} [\![W, D]\!]_t$	IC(t)	Extension
$\delta_1\delta_2$	$O_0(q \wedge r) \wedge O_1(q \wedge r \wedge p) \wedge O_2(q \wedge r \wedge p)$	$\wedge\ \top$	Th$\{p, q, r\}$
$\delta_2\delta_1$	$O_0(q \wedge r) \wedge O_1(q \wedge r \wedge \neg p) \wedge O_2(q \wedge r \wedge \neg p)$	$\wedge\ \top$	Th$\{\neg p, q, r\}$

If we let $\delta_1 < \delta_2$, then $\delta_2\delta_1 \notin T$, thus the only extension is Th$\{p, q, r\}$. Similarly if we let $\delta_2 < \delta_1$, the only extension is Th$\{\neg p, q, r\}$. If the order is empty, the representation entails $O_2(q \wedge r \wedge p) \vee O_2(q \wedge r \wedge \neg p)$, which is the Æ$_\top$-representation of the corresponding two extensions.

3.2 Adequacy of Translation

The proof that the translation determines all and only order-preserving extensions follows straightforwardly from the previous results in the paper.

Lemma 7. *Let t be an enumeration of D. Then $\bigwedge_{s \preceq t} [\![W, D]\!]_s \wedge IC(t)$ is consistent in Æ$_\top$ iff t is a felicitous enumeration of D.*

Proof. Since IC(t) formalizes the condition for t being felicitous, it is immediate that $\bigwedge_{s \preceq t} O_{|s|}\psi_s$ and IC(t) are true in the same model iff t is felicitous. The result follows from this and Lemma 6. □

Lemma 8. *The extensions of* (W, D) *are the sets* $Th(\psi_t)$ *for enumerations* t *of* D *such that* $\bigwedge_{s \preceq t} [\![W, D]\!]_s \wedge IC(t)$ *is consistent in* $\text{Æ}_\top$.

Proof. By Lemma 2 and Lemma 7. □

Theorem 1. *Let* (W, D) *be a default theory,* $n = |D|$*, and suppose for some* m

$$\vdash [\![W, D, <]\!] \equiv \bigvee_{0 \leq j \leq m} \bigwedge_{0 \leq i \leq n} O_i \varphi_i^j \ ,$$

where all φ_i^j *are purely Boolean and each disjunct* $\bigwedge_{0 \leq i \leq n} O_i \varphi_i^j$ *is* $\text{Æ}_\top$*-consistent. Then* $\{Th(\varphi_n^j) \mid 0 \leq j \leq m\}$ *is the set of extensions of* $(W, D, <)$.

Proof. By Lemma 4 and Lemma 8. □

3.3 Alternative Translations

Prior attempts to relate default logic and autoepistemic logic gave rise to a variety of mappings from default rules to autoepistemic formulae. It has been a general conception that the different mappings allow different sets of extensions to be generated [4], and therefore that the choice of mapping would determine the degree of correspondence one would find.

We now show that some mappings turn out to be equivalent in our encoding. This is due to the integrity constraint, which controls the impact a default may have on possible extensions. Such equivalences would seem to indicate that the differences between mappings are not as substantial as argued in the literature.

The mapping tr used in the encoding is identical to that of the HAEL system [10]. We will study two mappings that are simple modifications of tr, defined by

$$\text{tr}_1(r\delta) = (\alpha \wedge B_{|r|}\alpha \wedge b_{|r|}\beta) \supset \gamma \text{ and}$$
$$\text{tr}_2(r\delta) = (\alpha \wedge B_{|r\delta|}\alpha \wedge b_{|r\delta|}\beta) \supset \gamma \ , \text{ respectively.}$$

We now proceed to show that the lemmata of the previous section also hold for these two translations. Write $[\![W, D]\!]_s^1$ and $[\![W, D]\!]_s^2$ for the two alternative translation functions; we show that

$$\vdash \bigwedge_{s \preceq t} [\![W, D]\!]_s \wedge \text{IC}(t) \equiv \bigwedge_{s \preceq t} [\![W, D]\!]_s^1 \wedge \text{IC}(t) \equiv \bigwedge_{s \preceq t} [\![W, D]\!]_s^2 \wedge \text{IC}(t)$$

for any enumeration t of D. To see that the first equivalence holds, observe that a proof of Lemma 6 for the translation $[\![W, D]\!]_s^1$ would proceed precisely as the above proof of this lemma, until it remains to prove

$$M \vDash O_{|s\delta|}(\psi_s \wedge ((\alpha \wedge B_{|s|}\alpha \wedge b_{|s|}\beta) \supset \gamma)) \equiv O_{|s\delta|}\psi_{s\delta}$$

under the assumption that $M \vDash O_{|s|}\psi_s$. Now if $M \vDash B_{|s|}\alpha$, then $\psi_s \vdash \alpha$, hence

$$M \vDash O_{|s\delta|}(\psi_s \wedge ((\alpha \wedge B_{|s|}\alpha \wedge b_{|s|}\beta) \supset \gamma)) \equiv O_{|s\delta|}(\psi_s \wedge ((B_{|s|}\alpha \wedge b_{|s|}\beta) \supset \gamma)).$$

The rest of the proof then proceeds as originally. The equivalence between the two latter translations can be seen from the fact that for any φ both translations imply $B_{|r|}\varphi \supset B_{|s|}\varphi$ and $b_{|s|}\varphi \supset b_{|r|}\varphi$ whenever $r \preceq s$, hence in combination with IC(t) either translation implies the equivalence between tr_1 and tr_2.

Notice that the mapping tr_2 is the one adopted by Chen [4]. He introduces this mapping in order find an autoepistemic correspondence of default theories (W, D) where W and the prerequisites of all $\delta \in D$ are conjunctions of literals. With this mapping, Lemma 8 shows the correspondence to *all* default theories.

4 Discussion and Future Work

This article presents a method for representing ordered default theories in modal logic by encoding these into formulae in Æ. We have shown that a classical default theory may be treated as a special case of an ordered default theory, and thus that the encoding is adequate for representing classical default reasoning. These results open up a number of possibilities for using the ideas and the toolbox of modal logic for interpreting reasoning with preference.

The idea of having different steps in the generation of an extension correspond to distinct modalities has strong intuitive value in the sense that it provides support for the idea that the confidence in default beliefs are relative to the number of defaults that have been tested. To conclude something by default is one way of restricting the set of worlds or situations that one considers to be *plausible*, and thereby be subject to a greater risk of error. The representation of this idea does not have a counterpart in the framework of default logic.

There are many interesting variants and extensions of the encoding to consider. In section 3.3 above, alternative, *equivalent* translations were discussed, but there is also room for a discussion and comparison of translations corresponding to alternative notions of an ordered default theory. Furthermore we have, up until now, only dealt with the idea of static preference, i.e. preferences that are fixed in every circumstance. For the future, we plan to encompass the idea of dynamic preference as well. Many preferences are context independent, especially within the field of medical and legal reasoning. Dynamic preference may not be encoded in the current translation. Handling dynamic preferences presupposes that preferences may be subject to reasoning, thus we need to represent preferences at the object level. Making preferences objects of reasoning could provide many interesting perspectives on default reasoning.

The system Æ has been generalized to a multi-agent logic [18,19]. Understanding default reasoning in multi-agent contexts is a task that we will work on in the future. It would be fruitful to consider the idea of static preference also in the multi-agent case. This way the idea of an agent having beliefs about the preferences of other agents might be represented.

Another idea that we want to develop further is that of constraining reasoning by excluding some possible worlds from the set of conceivable worlds. This restriction would provide us with an even more fine-grained method for selecting the most plausible extensions. The Æ framework [20] is defined so that any

nonempty sets of points can serve as universe, allowing a flexible representation of the idea of personal necessity. Of course, such restrictions would go beyond the ideas of classical default logic.

References

1. Baader F., Hollunder B.: How to Prefer More Specific Defaults in Terminological Default Logic. Technical Report RR-92-58, DFKI, December 1993
2. Brewka, G.: Adding priorities and specificity to default logic. In *JELIA 1994, Proceedings, LNAI 838, Springer*, pages 247–260
3. Brewka, G., Eiter, T.: Prioritizing Default Logic. In St. Hölldobler, editor: Intellectics and Computational Logic. Papers in Honor of Wolfgang Bibel. Kluwer Academic Publisher, 2000
4. Chen, J.: The Logic of Only Knowing as a Unified Framework for Non-monotonic Reasoning. ISMIS '93, Proceedings, Lecture Notes in Computer Science **689** (1993) 152–161
5. Delgrande, J. P., Schaub, T.: Expressing Preferences in Default Logic. Artificial Intelligence **123** (2000) 41–87
6. Delgrande, J. P., Schaub, T., Tompits, H., Wang, K.: A Classification and Survey of Preference Handling Approaches in Nonmonotonic Reasoning. Computational Intelligence **20**(2) (2004) 308–334
7. Denecker, M., Marek, V. W., Truszczynski, M.: Uniform semantic treatment of default and autoepistemic logics. Artificial Intelligence **143** (2003) 79–122
8. Gottlob, G.: Complexity Results for Nonmonotonic Logics. Journal of Logic and Computation **2** (1992) 397–425
9. Gottlob, G.: Translating Default Logic into Standard Autoepistemic Logic. J. ACM 42(4) (1995) 711–740
10. Konolige, K.: Hierarchic Autoepistemic Theories for Nonmonotonic Reasoning. Pages 439–443 of *Proceedings of the Seventh National Conference on Artificial Intelligence (AAAI'88).* Morgan Kaufmann Publishers
11. Konolige, K.: On the relationship between default and autoepistemic logic Artificial Intelligence **35** (1988) 343–382
12. Levesque, H. J.: All I know: A study in autoepistemic logic. Artificial Intelligence **42** (1990) 263–309
13. Lian, E. H., Langholm, T., Waaler, A.: Only Knowing with Confidence Levels: Reductions and Complexity. In *JELIA 2004, Proceedings, LNAI 3229, Springer*, pages 500–512
14. Marek, V. W., M. Truszczynski, M.: Nonmonotonic logics; context dependent reasoning. Springer Verlag. 1993
15. Moore, R.: Semantical Considerations on Nonmonotonic Logic. Artificial Intelligence **25** (1985) 75–94
16. Reiter, R.: A logic for default reasoning. Artificial Intelligence **13** (1980) 81–132
17. Rintanen, J.: Prioritized autoepistemic logic. In *JELIA 1994, Proceedings, LNAI 838, Springer*, pages 232–246
18. Waaler, A.: Consistency proofs for systems of multi-agent only knowing. To appear in Advances in Modal Logic, Volume 5.
19. Waaler, A., Solhaug, B.: Semantics for multi-agent only knowing (extended abstract). In *Proceedings of TARK X* (2005), ACM Digital Library, pages 109–125
20. Waaler, A., Klüwer, J. W., Langholm, T., Lian, E.: Only Knowing with Degrees of Confidence. To appear in the *Journal of Applied Logics.*
with Degrees of Confidence. Submitted for publication.

A Social Semantics for Multi-agent Systems*

Francesco Buccafurri and Gianluca Caminiti

DIMET, Università degli Studi Mediterranea di Reggio Calabria,
via Graziella, loc. Feo di Vito, I-89060 Reggio Calabria, Italy
{bucca, gianluca.caminiti}@unirc.it

Abstract. As in human world many of our goals could not be achieved without interacting with other people, in case many agents are part of the same environment one agent should be aware that he is not alone and he cannot assume other agents sharing his own goals. Moreover, he may be required to interact with other agents and to reason about their mental state in order to find out potential friends to join with (or opponents to fight against). In this paper we focus on a language derived from logic programming which both supports the representation of mental states of agent communities and provides each agent with the capability of reasoning about other agents' mental states and acting accordingly. The proposed semantics is shown to be translatable into stable model semantics of logic programs with aggregates.

1 Introduction

Beside *autonomy*, agents [16,15] may be required to have *social ability*, which is the capability of interacting with other self-interested agents and, as a consequence, producing *beliefs*, *desires* and *intentions* (BDI) [2,3] which may be dependent on such interactions. Social ability means not only using a common language for agent communication. In this respect, KQML [13] and FIPA ACL [7], both based on the *speech act theory* by Cohen and Levesque [6], represent the main efforts done in the last years. Another important issue is reasoning about the content of such a communication [15,14,11].

In this paper we focus on a language derived from logic programming which both supports the representation of mental states of agent communities and provides each agent with the capability of reasoning about other agents' mental states and acting accordingly. Consider the following example: There are four agents which have been invited to the same wedding party. Some agents are less autonomous than the others, i.e. they may decide either to join the party or not to go at all, possibly depending on the other agents' choice. Moreover some agents may tolerate some options. These are the desires of the agents:

Agent$_1$ will go to the party only if at least the half of the total number of agents (not including himself) goes there.

* This work was partially supported by WASP (Working Group on Answer Set Programming) IST-2001-37004, 5th framework programme, Information Society Technologies, Action Line FET (Future and emerging technologies).

C. Baral et al. (Eds.): LPNMR 2005, LNAI 3662, pp. 317–329, 2005.
© Springer-Verlag Berlin Heidelberg 2005

Agent$_2$ possibly does not go to the party, but he tolerates such an option. In case he goes, then he possibly drives the car.

Agent$_3$ would like to join the party together with **Agent**$_2$, but he is not so much safe with **Agent**$_2$'s driving skill. Thus he decides to go to the party only if **Agent**$_2$ both goes there and does not want to drive the car.

Agent$_4$ does not go to the party.

It is possible to represent the above desires using logic programming with negation as failure (*not*) where each agent is represented by a single program and requested/desired items (representing the mental state of the agent) are modelled as atoms occurring inside rule heads. In particular, mandatory items are modelled as facts. Moreover, it is possible to represent tolerated items, i.e. items which are not requested, but possibly accepted. To this aim we use the predicate *okay*(), previously introduced in [5].

However, representing the requests/acceptances of single agents in a community is not enough. A social language should provide also a machinery to handle compromises among those agents. Thus, we introduce a new construct providing one agent with the ability to reason about other agents' mental state and then to act accordingly. Program rules may have the form:

$$head \leftarrow [selection_condition]\{body\}, \quad (1)$$

where *selection_condition* predicates about some social condition concerning either the cardinality of communities or particular individuals satisfying *body*.

For instance, consider the following rule, belonging to a program representing a given agent **A**: $\texttt{a} \leftarrow [\, l, h \,]\, \{\texttt{b},\ not\ \texttt{c}\}$. This rule means that **A** will require a only if n agents (other than **A**) exist such that they require or tolerate b, do not require or tolerate c and it holds that $0 \leq l \leq n \leq h \leq n_{agent} - 1$, where n_{agent} is the total number of agents[1].

This enriched language is referred to as *SOcial Logic Programming* (SOLP). The wedding party example above may be represented by the four SOLP programs shown in Table 1, where the program $\mathcal{P}_4$ is empty since the corresponding agent has not any desire to express.

The intended models must represent the mental states of each agent inside the community. For instance, the agents' choices w.r.t. the party can be: $\{\}$, $\{\texttt{go_wedding}_{\mathcal{P}_1}, \texttt{go_wedding}_{\mathcal{P}_2}, \texttt{drive}_{\mathcal{P}_2}\}$, and $\{\texttt{go_wedding}_{\mathcal{P}_1}, \texttt{go_wedding}_{\mathcal{P}_2}, \texttt{go_wedding}_{\mathcal{P}_3}\}$, where the subscript $_{\mathcal{P}_i}$ $(1 \leq i \leq n_{agent})$ references, for each atom in a model, the program (resp. agent) that atom is entailed by. The models respectively mean that either (*i*) no agent will go to the party, (*ii*) only **Agent**$_1$ and **Agent**$_2$ will go and also **Agent**$_2$ will drive the car, or (*iii*) all agents but **Agent**$_4$ will go to the party.

Indeed, **Agent**$_4$ anyway does not go. On the one hand, if **Agent**$_2$ does not go to the party, then **Agent**$_3$ will do the same. Now, let n' be the number of agents which are going to the party, it is $n' = 0$. **Agent**$_1$ requires that at

[1] By default, $l = 0$ and $h = n_{agent} - 1$.

Table 1. The wedding party example

$\mathcal{P}_1$ (**Agent**$_1$) :	$\mathcal{P}_2$ (**Agent**$_2$) :
$\texttt{go_wedding} \leftarrow [\frac{n_{agent}}{2} - 1,\ ]\{\texttt{go_wedding}\}$	$okay(\texttt{go_wedding}) \leftarrow$ $okay(\texttt{drive}) \leftarrow \texttt{go_wedding}$
$\mathcal{P}_3$ (**Agent**$_3$) :	$\mathcal{P}_4$ (**Agent**$_4$) :
$\texttt{go_wedding} \leftarrow [\textbf{Agent}_2]\{\texttt{go_wedding},$ $not\ \texttt{drive}\}$	*empty program*

least $\nu = \frac{n_{agent}}{2} - 1$ agents (other than himself) go to the party, but since it is $\nu = \frac{4}{2} - 1 = 1$ and $n' < \nu$, then **Agent**$_1$ does not go to the party (case (i)).

On the other hand, if **Agent**$_2$ goes to the party, it is possible that he wants either to drive the car or not. If he wants to drive, then **Agent**$_3$ will not join the party. Now, it is $n' = 1 = \nu$, then **Agent**$_1$ will go to the party (case (ii)). Otherwise, if **Agent**$_2$ does not want to drive the car, then all conditions required by **Agent**$_3$ are satisfied, thus he will go to the party. Now, it is $n' = 2 > \nu$ and then **Agent**$_1$ will join the party too (case (iii)).

The intended models are referred to as *social models*, since they express the results of the interactions among agents.

Our work is strongly related to [5], where the *Joint Fixpoint Semantics* (JFP), that is a semantics providing a way to reach a compromise (in terms of a common agreement) among many agents, is proposed. Therein, each model contains atoms representing items being common to all the agents. Our paper extends such a semantics, providing *feature-selective atom subset community*, i.e. given a set S of SOLP programs representing a community of agents, a program $\mathcal{P} \in S$ and a rule $r \in \mathcal{P}$ of the form $head \leftarrow [selection_condition]\{body\}$, then *head* will belong to an intended model if all properties enclosed in $\{body\}$ are entailed by either (i) any subset $S' \subseteq (S \setminus \{\mathcal{P}\})$ of programs with a given cardinality (specified by $[selection_condition]$) or (ii) some particular program different from $\mathcal{P}$.

An example of case (i) is shown in Table 1 by the program $\mathcal{P}_1$: An intended model M will include the atom $\texttt{go_wedding}_{\mathcal{P}_1}$ if a set of programs $S' \subseteq \{\mathcal{P}_2, \mathcal{P}_3, \mathcal{P}_4\}$ exists such that $\forall \mathcal{P}_i \in S'$, $\texttt{go_wedding}_{\mathcal{P}_i} \in M$ and $|S'| \geq \frac{n_{agent}}{2} - 1$. An example of case (ii) is represented by the program $\mathcal{P}_3$, which requests the atom $\texttt{go_wedding}_{\mathcal{P}_3}$ to be part of an intended model M if $\texttt{go_wedding}_{\mathcal{P}_2}$ belongs to M, but the atom $\texttt{drive}_{\mathcal{P}_2}$ does not.

Importantly, social constraints can be nested. Consider for example the program: $\texttt{download}(X) \leftarrow [min,\ ]\{\texttt{shared}(X), [1,\ ]\{not\ \texttt{incomplete}(X)\}\}, \texttt{file}(X)$. This program represents a Peer-to-Peer file-sharing system where a user can share his collection of files with other users on the Internet. In order to get better performances, a file is split into several parts being downloaded separately (possibly

each part from a different user)[2]. Thus, the program describes the behavior of an agent (acting on behalf of a given user) that wants to download any file X being shared by at least a number *min* of users such that at least one of them owns a complete version of X.

We show also that, given a set of SOLP programs in input, a source-to-source transformation is possible which provides as output a single DLP$^{\mathcal{A}}$ [8] program whose stable models are in one-to-one correspondence with the intended ones. We recall that DLP$^{\mathcal{A}}$ is basically disjunctive logic programming with aggregate functions, supported by the DLV system [9]. The translation to DLP$^{\mathcal{A}}$ give us the ability of exploiting DLV (widely accepted as the state-of-the-art system implementing disjunctive logic programming)[3]. Observe that since our language includes neither disjunction nor classical negation (even though the extensions to these cases could be considered), both disjunction and classical negation of DLP$^{\mathcal{A}}$ are never enabled by our translation. Moreover, Section 5 shows that our kind of social reasoning is not trivial, since even in the case of positive programs, the semantics of SOLP has a computational complexity which is NP complete.

The paper is organized as follows: in Sections 2 and 3 we respectively define the notion of SOLP programs and define their semantics (*Social Semantics*). In Section 4 we illustrate how a set of SOLP programs, each representing a different agent, is translated into a single DLP$^{\mathcal{A}}$ logic program whose stable models describe the mental states of the whole agent community and then we show that such a translation is correct. In Section 5 we prove that the Social Semantics extends the JFP Semantics [5] and we study the complexity of the problem of searching for a social model. In Section 6 we describe how this novel approach may be used for knowledge representation and finally, we draw our conclusions. For space restrictions, proofs of theorems and lemmata are omitted. They can be found in [4].

2 Social Logic Programs: Basic Definitions

In this section we introduce the notion of SOLP program.

A *term* is either a variable or a constant. An *atom* or *positive literal* is an expression $p(t_1, \cdots, t_n)$, where p is a *predicate* of arity n and $t_1, \cdots, t_n$ are terms. A *negative literal* is the *negation as failure (NAF) not a* of a given atom a.

Definition 1. A (n-)*social selection constraint s*, said also (n-)SSC, is an expression of the form $cond(s)$ $property(s)$, such that:

(1) $cond(s)$ is an expression $[\alpha]$ where α is either (i) a pair of integers l, h such that $0 \leq l \leq h \leq n-1$, or (ii) an integer belonging to $\{1, \cdots, n\}$ said *program identifier*[4].

---

[2] Among others, KaZaA, EDonkey, WinMX and BitTorrent are the most popular Internet P2P file-sharing systems exploiting such a feature.

[3] Of course, our approach may easily be adapted to other systems supporting cardinality constraints, such as Smodels.

[4] We will show, at the end of the section, that the program identifier uniquely identifies a program (i.e., an agent).

(2) $property(s) = content(s) \cup skel(s)$, where $content(s)$ is a non-empty set of literals and $skel(s)$ is a (possibly empty) set of SSCs.

Concerning item (1) of the above definition, in case (i), $cond(s)$ is said *cardinal selection condition*, while, in case (ii), $cond(s)$ is said *member selection condition*.

n-social selection constraints operate over a collection of n programs (we will formally define later in this section which kind of program are allowed). Thus, with a little abuse of notation, we often denote a member selection condition by $[\mathcal{P}_j]$ instead of $[j]$.

Concerning item (2) of Definition 1, if $skel(s) = \emptyset$ then s is said *simple*. For a simple SSC s such that $content(s)$ is singleton, the enclosing braces can be omitted. Finally, given a SSC s, the formula *not* s is said the NAF of s.

In our initial wedding party example, $[\frac{n_{agent}}{2} - 1,]\{\texttt{go_wedding}\}$ and $[\mathbf{Agent}_2]$ $\{\texttt{go_wedding}, \mathit{not}\, \texttt{drive}\}$ are two simple SSCs. On the contrary, the SSC occurring in the example regarding a Peer-to-Peer system (see Page 319) is not simple.

As a further example, if $s = [l,h]\{a,b,c,[l_1,h_1]\{d,[l_2,h_2]e\},[l_3,h_3]f\}$, then s is not simple, $content(s) = \{a,b,c\}$ and $skel(s) = \{[l_1,h_1]\{d,[l_2,h_2]e\},[l_3,h_3]f\}$.

Now we define a function which returns, for a given SSC s, its nesting depth.

Given a SSC s, we define the function *depth* as follows:

$$\begin{cases} depth(s) = depth(s') + 1, & \text{if } \exists s' \mid s \in skel(s') \\ depth(s) = 0, & \text{otherwise.} \end{cases}$$

Given two SSCs s and s' such that $cond(s) = [l,h]$ and $cond(s') = [l',h']$, i.e. they are cardinal selection conditions, we say that $cond(s') \subseteq cond(s)$ if $h' \leq h$.

A SSC s is *well-formed* if either (i) s is simple, or (ii) s is not simple, $cond(s)$ is a cardinal selection condition and $\forall s' \in skel(s)$ it holds that:

(a) If $cond(s')$ is a cardinal selection condition, then s' is *well-formed* and $cond(s') \subseteq cond(s)$;
(b) If $cond(s')$ is a member selection condition, then s' is simple.

From now on, we consider only well-formed SSCs.

Example 1. The SSC $s = [1,8]\{a,[3,6]\{b,[\mathbf{Agent_X}]\{c,d\}\}\}$ is well-formed, while $s_1 = [4,7]\{a,[3,9]b\}$ is not a well-formed SSC, because $[3,9] \not\subseteq [4,7]$.

We introduce now the notion of rule. Our definition generalizes the notion of classical logic rule.

Definition 2. A $(n\text{-})$*social rule* r is a is a formula $a \leftarrow b_1 \wedge \cdots \wedge b_m \wedge s_1 \wedge \cdots \wedge s_k$ $(m \geq 0,\ k \geq 0)$, where a is an atom, each b_i $(1 \leq i \leq m)$ is a literal and each s_j $(1 \leq j \leq k)$ is either a n-SSC or the NAF of a n-SSC. The atom a is said the *head* of r, while the conjunction $b_1 \wedge \cdots \wedge b_m \wedge s_1 \wedge \cdots \wedge s_k$ is said the *body* of r. In case a is of the form $okay(p)$, where p is an atom, then r it is said $(n\text{-})$*tolerance (social) rule* and p is said the *head* of r. In case $k = 0$, a social non-tolerance rule is said *classical rule*.

Given a rule r, we denote by $head(r)$ (resp. $body(r)$) the head (resp. the body) of r. Moreover, r is said a *fact* in case the body is empty, while r is said an *integrity constraint* if the head is missing.

Definition 3. A SOLP *collection* is a set $\{\mathcal{P}_1, \cdots, \mathcal{P}_n\}$ of SOLP programs, where each SOLP *program* is a set of n-social rules. The cardinal i $(1 \leq i \leq n)$ is called *program identifier* of the program $\mathcal{P}_i$.

A SOLP program is *positive* if no NAF symbol *not* occurs in it. For the sake of presentation we only refer, in the following sections, to *ground* (i.e., variable-free) SOLP programs – the extension to the general case is straightforward.

3 Semantics of SOLP Programs

In this section we introduce the *Social Semantics*, i.e. the semantics of a collection of SOLP programs. We assume the reader is familiar with the basic concepts of logic programming [1,12].

We start by introducing the notion of interpretation for a single SOLP program (note that this is the same as for classical programs). An *interpretation* for a ground SOLP program $\mathcal{P}$ is a subset of $Var(\mathcal{P})$, where $Var(\mathcal{P})$ is the set of atoms appearing in $\mathcal{P}$. A positive literal a (resp. a negative literal *not* a) is *true* w.r.t. an interpretation I if $a \in I$ (resp. $a \notin I$); otherwise it is *false*. A rule is *satisfied* (or is *true*) w.r.t. I if its head is true or its body is false w.r.t. I.

Before defining the intended models of our semantics, we need some preliminary definitions. Let $\mathcal{P}$ be a SOLP program. We define the *autonomous reduction* of $\mathcal{P}$, denoted by $A\mathcal{P}$, the program obtained from $\mathcal{P}$ by removing all the SSCs from the rules in $\mathcal{P}$. Thus, if the program $\mathcal{P}$ represents the social behavior of an agent, then $A\mathcal{P}$ represents the behavior of the same agent in case he decides to operate independently of the other agents.

Given a SOLP program $\mathcal{P}$ and an interpretation $I \subseteq Var(A\mathcal{P})$, let $\mathrm{CL}(A\mathcal{P})$ (resp. $\mathrm{TR}(A\mathcal{P})$) be the set of classical (resp. tolerance) rules in $A\mathcal{P}$. The *autonomous immediate consequence operator* $AT_{\mathcal{P}}$ is the function from $2^{Var(A\mathcal{P})}$ to $2^{Var(A\mathcal{P})}$ defined as follows:

$$\begin{aligned} AT_{\mathcal{P}}(I) = & \{head(r) \mid \forall r \in CL(A\mathcal{P}), body(r) \text{ is true w.r.t. } I\} \cup \\ & \{head(r) \mid \forall r \in TR(A\mathcal{P}), body(r) \wedge head(r) \text{ is true w.r.t. } I\}. \end{aligned}$$

Definition 4. An interpretation I for a SOLP program $\mathcal{P}$ is an *autonomous fixpoint* of $\mathcal{P}$ if I is a fixpoint of the associated transformation $AT_{\mathcal{P}}$, i.e. if $AT_{\mathcal{P}}(I) = I$. The set of all autonomous fixpoints of $\mathcal{P}$ is denoted by $AFP(\mathcal{P})$.

Thus, the autonomous fixpoints of a given SOLP program $\mathcal{P}$ represent the mental states of the corresponding agent, whenever every social constraint in $\mathcal{P}$ is discarded.

Definition 5. Let $\mathcal{P}$ be a SOLP program and L be a set of literals. The *labeled version* of L w.r.t. $\mathcal{P}$ is the set $L_{\mathcal{P}} = \{a_{\mathcal{P}} \mid a \in L\}$.

Let $C = \{\mathcal{P}_1, \cdots, \mathcal{P}_n\}$ be a SOLP collection. A *social interpretation* for C is a set $\bar{I} = I^1_{\mathcal{P}_1} \cup \cdots \cup I^n_{\mathcal{P}_n}$, where I^j is an interpretation for $\mathcal{P}_j$ $(1 \leq j \leq n)$.

Example 2. If $C = \{\mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3\}$, $I^1 = \{a, b, c\}$, $I^2 = \{a, d, e\}$ and $I^3 = \{b, c, d\}$, where I^j is an interpretation for $\mathcal{P}_j$ $(1 \leq j \leq 3)$, then $\bar{I} = \{a_{\mathcal{P}_1}, b_{\mathcal{P}_1}, c_{\mathcal{P}_1}, a_{\mathcal{P}_2}, d_{\mathcal{P}_2}, e_{\mathcal{P}_2}, b_{\mathcal{P}_3}, c_{\mathcal{P}_3}, d_{\mathcal{P}_3}\}$ is a social interpretation for C.

Let $C = \{\mathcal{P}_1, \cdots, \mathcal{P}_n\}$ be a SOLP collection and $\mathcal{P} \in C$. Given a social interpretation $\bar{I}$ for C, a positive literal $a \in Var(\mathcal{P})$ (resp. a negative literal *not* a) is *true* w.r.t. $\bar{I}$ if $a_{\mathcal{P}} \in \bar{I}$ (resp. $a_{\mathcal{P}} \notin \bar{I}$); otherwise it is *false*.

Before giving the definition of truth for a SSC, we introduce a way to reference any SSC s (and also every SSC nested in s) occurring in a given rule r of a SOLP program $\mathcal{P}$.

Given a SOLP program $\mathcal{P}$, a social rule $r \in \mathcal{P}$ and an integer $n \geq 0$, we define the set $MSSC^{\langle \mathcal{P}, r, n \rangle} = \{s \mid s \text{ is a SSC occurring in } r \in \mathcal{P} \wedge depth(s) = n\}$. Observe that $MSSC^{\langle \mathcal{P}, r, 0 \rangle}$ denotes the set of SSCs as they appear in the rule r of the SOLP program $\mathcal{P}$.

Example 3. Let $a \leftarrow [1, 8]\{a, [3, 6]\{b, [\mathbf{Agent_X}]\{c, d\}\}\}, [2, 3]\{e, f\}$ be a rule r in a SOLP program $\mathcal{P}$. Then:

$$MSSC^{\langle \mathcal{P}, r, 0 \rangle} = \{\ [1, 8]\{a, [3, 6]\{b, [\mathbf{Agent_X}]\{c, d\}\}\}, [2, 3]\{e, f\}\ \},$$
$$MSSC^{\langle \mathcal{P}, r, 1 \rangle} = \{\ [3, 6]\{b, [\mathbf{Agent_X}]\{c, d\}\}\ \},$$
$$MSSC^{\langle \mathcal{P}, r, 2 \rangle} = \{\ [\mathbf{Agent_X}]\{c, d\}\ \},$$
$$MSSC^{\langle \mathcal{P}, r, 3 \rangle} = \emptyset.$$

Given a SOLP program $\mathcal{P}$, we define the set $MSSC^{\mathcal{P}} = \bigcup_{r \in \mathcal{P}} MSSC^{\langle \mathcal{P}, r, 0 \rangle}$.

Thus $MSSC^{\mathcal{P}}$ is the set of all the SSCs (with depth 0) occurring in $\mathcal{P}$.

Now we provide the definition of truth of a SSC w.r.t. a given social interpretation and, subsequently, the definition of truth of a social rule.

Definition 6. Let $C = \{\mathcal{P}_1, \cdots, \mathcal{P}_n\}$ be a SOLP collection, $C' \subseteq C$ and $\mathcal{P}_j \in C'$. Given a social interpretation $\bar{I}$ for C' and a n-SSC $s \in MSSC^{\mathcal{P}_j}$, we say that s is *true* in C' w.r.t. $\bar{I}$ if it holds that either:

(1) $cond(s) = [k] \wedge$
$\exists \mathcal{P}_k \in C' \mid \forall a_{\mathcal{P}_k} \in (content(s))_{\mathcal{P}_k} \quad a$ is true w.r.t. $\bar{I}$, or

(2) $cond(s) = [l, h] \wedge$
$\exists D \subseteq C' \setminus \{\mathcal{P}_j\} \mid l \leq |D| \leq h \wedge$
$\forall a_{\mathcal{P}} \in \bigcup_{\mathcal{P} \in D} (content(s))_{\mathcal{P}} \quad a$ is true w.r.t. $\bar{I} \wedge$
$\forall s' \in skel(s)\ \exists D' \subseteq D \mid s'$ is true in D' w.r.t. $\bar{I}$,

where l, h and k are integers (observe that k is a program identifier). If $C' = C$, then we simply say that s is *true* w.r.t. $\bar{I}$. A n-SSC not true (in C') w.r.t. $\bar{I}$ is said *false* (in C') w.r.t. $\bar{I}$.

Finally, the NAF of a n-SSC s, *not* s, is said *true* (resp. *false*) (in C') w.r.t. $\bar{I}$ if s is false (resp. true) (in C') w.r.t. $\bar{I}$.

Thus, given a SSC s included in $\mathcal{P}_j$, s is true w.r.t. a social interpretation $\bar{I}$ if a single SOLP program corresponding to a program identifier k (resp. a set of

SOLP programs) exists (resp. not including $\mathcal{P}_j$) such that all the elements in *property*(s) are true w.r.t. $\bar{I}$. Observe that the truth of *property*(s) w.r.t. $\bar{I}$ is possibly defined recursively, since s may contain nested SSCs.

Once the notion of truth of SSCs has been defined, we are able to define the notion of satisfaction of a social rule w.r.t. a social interpretation.

Let $C = \{\mathcal{P}_1, \cdots, \mathcal{P}_n\}$ be a SOLP collection and $\mathcal{P} \in C$. Given a social interpretation $\bar{I}$ for C, a social rule in $\mathcal{P}$ is *satisfied* (or is *true*) w.r.t. $\bar{I}$ if its head is true w.r.t. $\bar{I}$ or its body is false w.r.t. $\bar{I}$.

Given a SOLP collection $\{\mathcal{P}_1, \cdots, \mathcal{P}_n\}$, we define the set of *candidate social interpretations* for $\mathcal{P}_1, \cdots, \mathcal{P}_n$ as

$$\mathcal{U}(\mathcal{P}_1, \cdots, \mathcal{P}_n) = \left\{ F^1_{\mathcal{P}_1} \cup \cdots \cup F^n_{\mathcal{P}_n} \mid F^i \in AFP(\mathcal{P}_i), 1 \leq i \leq n \right\}.$$

where, recall, $AFP(\mathcal{P}_i)$ is the set of autonomous fixpoints of the SOLP program $\mathcal{P}_i$, introduced in Definition 4 and by $F^i_{\mathcal{P}}$ $(1 \leq i \leq n)$ we denote the labeled version of F^i w.r.t. $\mathcal{P}$ (see Definition 5). $\mathcal{U}(\mathcal{P}_1, \cdots, \mathcal{P}_n)$ represents all the configurations obtained by combining the autonomous (i.e. without considering the social constraints) mental states of the agents corresponding to the programs $\mathcal{P}_1, \cdots, \mathcal{P}_n$. Each candidate social interpretation is a candidate intended model.

The intended models are then obtained by enabling the social constraints.

Now, we are ready to give the definition of intended model w.r.t. the Social Semantics.

Definition 7. Given a SOLP collection $C = \{\mathcal{P}_1, \cdots, \mathcal{P}_n\}$, a candidate social interpretation $\bar{I}$ for C is a *social model* of C if $\forall r \in \bigcup_{1 \leq i \leq n} \mathcal{P}_i$, r is true w.r.t. $\bar{I}$.

Definition 8. Given a SOLP collection $\{\mathcal{P}_1, \cdots, \mathcal{P}_n\}$, the *Social Semantics* (often referred to as *S-Semantics*) of $\mathcal{P}_1, \cdots, \mathcal{P}_n$ is the set

$$\mathcal{SOS}(\mathcal{P}_1, \cdots, \mathcal{P}_n) = \{\bar{M} \mid \bar{M} \in \mathcal{U}(\mathcal{P}_1, \cdots, \mathcal{P}_n) \wedge \bar{M} \text{ is a social model of } \mathcal{P}_1, \cdots, \mathcal{P}_n\},$$

Thus $\mathcal{SOS}(\mathcal{P}_1, \cdots, \mathcal{P}_n)$ is the set of all social models of $\mathcal{P}_1, \cdots, \mathcal{P}_n$.

Given a SOLP collection $C = \{\mathcal{P}_1, \cdots, \mathcal{P}_n\}$ and a SOLP program $\mathcal{P} \in C$, we define the *S-Semantics* of $\mathcal{P}$ as

$$\mathcal{S}(\mathcal{P}) = \{F \mid F \in AFP(\mathcal{P}) \wedge \exists \bar{M} \in \mathcal{SOS}(\mathcal{P}_1, \cdots, \mathcal{P}_n) \mid F_{\mathcal{P}} \subseteq \bar{M}\},$$

Hence $\mathcal{S}(\mathcal{P})$ represents the autonomous mental states of an agent, corresponding to a SOLP program $\mathcal{P}$, which are also included in some social model of $\mathcal{P}_1, \cdots, \mathcal{P}_n$, and then fulfill all social requirements.

4 Translation

In this section we give the translation from SOLP under the Social Semantics to DLP$^{\mathcal{A}}$ [8] under Stable Model Semantics. We assume that the reader is familiar with the Stable Model Semantics [10]. Given a classical logic program $\mathcal{P}$, we denote by $SM(\mathcal{P})$ the set of all the stable models of $\mathcal{P}$.

Given a SOLP program $\mathcal{P}$, we define the set $USSC^{\mathcal{P}} = \bigcup_{r\in\mathcal{P}} \bigcup_{n\geq 0} MSSC^{\langle \mathcal{P},r,n\rangle}$. Thus, $USSC^{\mathcal{P}}$ includes all the SSCs (at any nesting depth) in $\mathcal{P}$.

Given a SOLP program $\mathcal{P}$, we define the functions ρ and g, each establishing a one-to-one correspondence between each element in $USSC^{\mathcal{P}}$ and a set of atoms L such that both (i) $L \cap Var(\mathcal{P}) = \emptyset$ and (ii) $\forall s,t \in USSC^{\mathcal{P}}, \rho(s) \neq g(t)$. Thus, given a SSC s included in a SOLP program $\mathcal{P}$, $\rho(s)$ is a unique positive literal identifying s and we denote by $(\rho(s))_{\mathcal{P}}$ the labeled version of $\rho(s)$ w.r.t. $\mathcal{P}$. Similar considerations hold for $g(s)$. Moreover, with a little abuse of notation we write $(g(s))_{\mathcal{P}}(x)$ denoting the labeled version of the predicate $(g(s))(x)$ w.r.t. $\mathcal{P}$.

Now we introduce the translation of a single SSC and then we extend such a translation to all SSCs included in a social rule, a SOLP program and a SOLP collection, respectively.

Definition 9. Given a SOLP collection $\{\mathcal{P}_1, \cdots, \mathcal{P}_j, \cdots, \mathcal{P}_n\}$ and $s \in USSC^{\mathcal{P}_j}$, we define the *translation of* s as the DLP$^{\mathcal{A}}$ program $\Psi^{\mathcal{P}_j}(s) = GUESS^{\mathcal{P}_j}(s) \cup CHECK^{\mathcal{P}_j}(s)$, where $GUESS^{\mathcal{P}_j}(s) =$

$$= \begin{cases} \{(g(s))_{\mathcal{P}_j}(k) \leftarrow \bigwedge_{b \in content(s)} b_{\mathcal{P}_k}\}, & \text{if } cond(s) = [k], \\ \{(g(s))_{\mathcal{P}_j}(i) \leftarrow \bigwedge_{b \in content(s)} b_{\mathcal{P}_i} \wedge \\ \qquad \bigwedge_{s' \in skel(s)} (g(s'))_{\mathcal{P}_j}(i) \mid \\ \qquad 1 \leq i \neq j \leq n\} \cup \\ \qquad \{GUESS^{\mathcal{P}_j}(s') \mid s' \in skel(s)\}, & \text{if } cond(s) = [l,h], \end{cases}$$

and $CHECK^{\mathcal{P}_j}(s) =$

$$= \begin{cases} \{(\rho(s))_{\mathcal{P}_j} \leftarrow (g(s))_{\mathcal{P}_j}(k)\}, & \text{if } cond(s) = [k], \\ \{(\rho(s))_{\mathcal{P}_j} \leftarrow l \leq \#\texttt{count}\{K : (g(s))_{\mathcal{P}_j}(K), K \neq j\} \leq h\} \cup \\ \qquad \{CHECK^{\mathcal{P}_j}(s') \mid s' \in skel(s)\}, & \text{if } cond(s) = [l,h], \end{cases}$$

where `#count` is an aggregate function which returns the cardinality of a set of literals satisfying some conditions [8]. Observe that the above translation produces a safe aggregate-stratified DLP$^{\mathcal{A}}$ program and thus the computational complexity remains the same as for standard DLP [8,9].

Given a SOLP program $\mathcal{P}_j$ and a social rule $r \in \mathcal{P}_j$, we define the *SSC translation of* r as the DLP$^{\mathcal{A}}$ program $T^{\mathcal{P}_j}(r) = \bigcup_{s \in MSSC^{\langle \mathcal{P},r,0\rangle}} \Psi^{\mathcal{P}_j}(s)$. Observe that, for any classical rule $r \in \mathcal{P}_j$, it holds that $T^{\mathcal{P}_j}(r) = \emptyset$.

Given a SOLP program $\mathcal{P}_j$, the *SSC translation of* $\mathcal{P}_j$ is the DLP$^{\mathcal{A}}$ program $W^{\mathcal{P}_j} = \bigcup_{r \in \mathcal{P}_j} T^{\mathcal{P}_j}(r)$.

Definition 10. Given a SOLP collection $\{\mathcal{P}_1, \cdots, \mathcal{P}_n\}$, we define the *SSC translation of the collection* as the DLP$^{\mathcal{A}}$ program $C(\mathcal{P}_1, \cdots, \mathcal{P}_n) = \bigcup_{1 \leq i \leq n} W^{\mathcal{P}_i}$.

Thus $C(\mathcal{P}_1, \cdots, \mathcal{P}_n)$ is a DLP$^{\mathcal{A}}$ program representing the translation of all the SSCs included in $\mathcal{P}_1, \cdots, \mathcal{P}_n$.

We have defined above the translation for the SSCs included in a SOLP program. Now we give the translation for the whole SOLP program, but first we need a preliminary processing of all tolerance rules. The latter is done by means of the transformation defined as follows:

Given a SOLP program $\mathcal{P}$, $\hat{\mathcal{P}} = \mathcal{P} \setminus TR(\mathcal{P}) \cup \{head(r) \leftarrow head(r) \wedge body(r) \mid r \in TR(\mathcal{P})\}$. Thus $\hat{\mathcal{P}}$ is obtained from $\mathcal{P}$ by replacing each tolerance rule $okay(p) \leftarrow body$ with the rule $p \leftarrow p, body$.

Definition 11. Let $\mathcal{P}$ be a SOLP program. We define the program $\Gamma'(\hat{\mathcal{P}})$ over the set of atoms $Var(\Gamma'(\hat{\mathcal{P}})) = \{a_{\mathcal{P}} \mid a \in Var(A\hat{\mathcal{P}})\} \cup \{a'_{\mathcal{P}} \mid a \in Var(A\hat{\mathcal{P}})\} \cup \{sa_{\mathcal{P}} \mid a \in Var(A\hat{\mathcal{P}})\} \cup \{fail_{\mathcal{P}}\}$ as $\Gamma'(\hat{\mathcal{P}}) = S'_1(\hat{\mathcal{P}}) \cup S'_2(\hat{\mathcal{P}}) \cup S'_3(\hat{\mathcal{P}})$, where $S'_1(\hat{\mathcal{P}})$, $S'_2(\hat{\mathcal{P}})$ and $S'_3(\hat{\mathcal{P}})$ are defined as follows:

$$
\begin{aligned}
S'_1(\hat{\mathcal{P}}) &= \{a_{\mathcal{P}} \leftarrow not\ a'_{\mathcal{P}} \mid a \in Var(A\hat{\mathcal{P}})\} \cup \{a'_{\mathcal{P}} \leftarrow not\ a_{\mathcal{P}} \mid a \in Var(A\hat{\mathcal{P}})\},\\
S'_2(\hat{\mathcal{P}}) &= \{sa_{\mathcal{P}} \leftarrow b^1_{\mathcal{P}}, \cdots, b^n_{\mathcal{P}}, (\rho(s_1))_{\mathcal{P}}, \cdots (\rho(s_m))_{\mathcal{P}} \mid a \leftarrow b_1, \cdots b_n, s_1, \cdots, s_m \in \mathcal{P}\},\\
S'_3(\hat{\mathcal{P}}) &= \{fail_{\mathcal{P}} \leftarrow not\ fail_{\mathcal{P}}, sa_{\mathcal{P}}, not\ a_{\mathcal{P}} \mid a \in Var(A\hat{\mathcal{P}})\} \cup\\
&\quad\ \{fail_{\mathcal{P}} \leftarrow not\ fail_{\mathcal{P}}, a_{\mathcal{P}}, not\ sa_{\mathcal{P}} \mid a \in Var(A\hat{\mathcal{P}})\}.
\end{aligned}
$$

Definition 12. Given a SOLP collection $\{\mathcal{P}_1, \cdots, \mathcal{P}_n\}$, we define the set $P'_u = \bigcup_{1 \le i \le n} \Gamma'(\hat{\mathcal{P}}_i)$.

Thus P'_u is a classical logic program representing the translation of the SOLP collection. This program is used in conjunction with the DLP$^{\mathcal{A}}$ program $C(\mathcal{P}_1, \cdots, \mathcal{P}_n)$ in order to enable the social constraints.

In the next theorem we state that a one-to-one correspondence exists between the social models in $SOS(\mathcal{P}_1, \cdots, \mathcal{P}_n)$ and the stable models of the DLP$^{\mathcal{A}}$ program $P'_u \cup \bar{C}$. First, we need the following definition and results:

Let $\mathcal{P}$ be a SOLP program and $M \subseteq Var(\mathcal{P})$. We denote by $[M]_{\mathcal{P}}$ the set $\{a_{\mathcal{P}} \mid a \in M\} \cup \{a'_{\mathcal{P}} \mid a \in Var(\mathcal{P}) \setminus M\} \cup \{sa_{\mathcal{P}} \mid a \in M\}$.

Lemma 1. Given a SOLP collection $SP = \{\mathcal{P}_1, \cdots, \mathcal{P}_n\}$, a social interpretation $\bar{I}$ for SP, a SOLP program $\mathcal{P}_j \in SP$ and a SSC $s \in MSSC^{\mathcal{P}_j}$, assume $\bar{C} = C(\mathcal{P}_1, \cdots, \mathcal{P}_n)$ and $Q = \{a \leftarrow \mid a \in \bar{I}\}$. Then:

$$s \text{ is true w.r.t. } \bar{I} \text{ iff } \exists M \in SM(\bar{C} \cup Q) \mid (\rho(s))_{\mathcal{P}_j} \in M.$$

Definition 13. Given a SOLP collection $SP = \{\mathcal{P}_1, \cdots, \mathcal{P}_n\}$, a social interpretation $\bar{I}$ for SP, a SOLP program $\mathcal{P}_j \in SP$ and a SSC $s \in MSSC^{\mathcal{P}_j}$, assume $\bar{C} = C(\mathcal{P}_1, \cdots, \mathcal{P}_n)$ and $Q = \{a \leftarrow \mid a \in \bar{I}\}$. We define the set

$$SAT_{\bar{I}}^{\mathcal{P}_j}(s) = \{\ h \mid h = head(r), r \in \Psi^{\mathcal{P}_j}(s) \wedge \exists M \in SM(\bar{C} \cup Q) \mid h \in M\}.$$

Thus, by virtue of Lemma 1, if s is true w.r.t. $\bar{I}$, then $SAT_{\bar{I}}^{\mathcal{P}_j}(s)$ includes the literal $(\rho(s))_{\mathcal{P}_j}$ and those heads of rules in $\Psi^{\mathcal{P}_j}(s)$ corresponding (by means of the functions ρ and g) to both s and the SSCs which are nested in s.

Theorem 1. Given a SOLP collection $\{\mathcal{P}_1, \cdots, \mathcal{P}_n\}$, and $\bar{C} = C(\mathcal{P}_1, \cdots, \mathcal{P}_n)$. Then $A = B$, where:

$$\begin{array}{ll} A = & SM(P'_u \cup \bar{C}) \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{and} \\ B = \{ & \bar{F} \cup \bar{G} \cup \bar{H} \mid \\ & \bar{F} = \bigcup_{1 \le i \le n} F^i_{\mathcal{P}_i} \wedge F^i \in AFP(\mathcal{P}_i) \wedge \bar{F} \in SOS(\mathcal{P}_1, \cdots, \mathcal{P}_n) \\ & \bar{G} = \bigcup_{1 \le i \le n} G^i_{\mathcal{P}_i} \wedge G^i_{\mathcal{P}_i} = [F^i]_{\mathcal{P}_i} \setminus F^i_{\mathcal{P}_i} \\ & \bar{H} = \bigcup_{1 \le i \le n} H^i_{\mathcal{P}_i} \wedge H^i_{\mathcal{P}_i} = \bigcup_{s \in MSSC^{\mathcal{P}_i}} SAT^{\mathcal{P}_i}_{\bar{F}}(s)\}. \end{array}$$

Thus each stable model $x \in A$ may be partitioned in three sets: $\bar{F}$ (the social model of the SOLP collection, which corresponds to x), $\bar{G}$ and $\bar{H}$ (both including overhead literals needed by the translation).

5 Social Models, Joint Fixpoints and Complexity

In this section we show that the Social Semantics extends the JFP semantics [5]. Basically, COLP programs are logic programs containing also *tolerance* rules, that are rules of the form $okay(p) \leftarrow body(r)$. The semantics of a collection of COLP programs is defined over classical programs obtained by the COLP programs by translating each rule of the form $okay(p) \leftarrow body(r)$ into the rule $p \leftarrow p, body(r)$. The semantics of a collection $\mathcal{P}_1, \cdots, \mathcal{P}_n$ of COLP programs is defined in [5] in terms of joint (i.e., common) fixpoints (of the *immediate consequence operator*) of the logic programs obtained from $\mathcal{P}_1, \cdots, \mathcal{P}_n$ by transforming *tolerance* rules occurring in them (as shown above). Recall that the *immediate consequence operator* $T_{\mathcal{P}}$ is a function from $2^{Var(\mathcal{P})}$ to $2^{Var(\mathcal{P})}$ defined as follows. For each interpretation $I \subseteq Var(\mathcal{P})$, $T_{\mathcal{P}}(I)$ is the set of all heads of rules in $\mathcal{P}$ whose bodies are true w.r.t. I.

First, we define a translation from COLP programs [5] to SOLP programs:

Definition 14. Given a COLP program $\mathcal{P}$ and an integer $n \ge 1$, the *SOLP translation* of $\mathcal{P}$ is a SOLP program $\sigma^n(\mathcal{P}) = \{\sigma_{rule}(r) \mid r \in \mathcal{P}\}$, where

$$\sigma_{rule}(r) = \begin{cases} head(r) \leftarrow [n-1, n-1]head(r), body(r) & \text{if } r \text{ is a classical rule,} \\ okay(p) \leftarrow [n-1, n-1]p, body(r) & \text{if } head(r) = okay(p). \end{cases}$$

The next theorem states that the JFP semantics is a special case of the Social Semantics. $JFP(\mathcal{P}_1, \cdots, \mathcal{P}_n)$ denotes the set of the joint fixpoints of $\mathcal{P}_1, \cdots, \mathcal{P}_n$.

Theorem 2. Given $n \ge 1$, let $\mathcal{P}_1, \cdots, \mathcal{P}_n$ be COLP programs and $Q_1, \cdots, Q_n$ be SOLP programs such that $Q_i = \sigma^n(\mathcal{P}_i)$. Then:

$$\mathcal{SOS}(Q_1, \cdots, Q_n) = \left\{ \bigcup_{1 \le i \le n} F_{Q_i} \mid F \in JFP(\mathcal{P}_1, \cdots, \mathcal{P}_n) \right\}.$$

Now we introduce a relevant decision problem w.r.t. the Social Semantics and discuss its complexity. Observe that the analysis is done in case of positive programs. Indeed, the case of non positive programs is straightforward: Since it is NP complete to determine whether a *single* non-positive program has a fixpoint, it is easy to see that the same holds for non-positive SOLP programs and autonomous fixpoints. Thus, checking whether a SOLP collection containing at least one non-positive SOLP program has a social model is trivially NP hard. Moreover, since this problem is easily seen to be in NP, it is NP complete.

PROBLEM SOS_n (Social Model Existence):

Instance: A SOLP collection $\mathcal{P}_1, \ldots, \mathcal{P}_n$
Question: Is $\mathcal{SOS}(\mathcal{P}_1, \ldots \mathcal{P}_n) \neq \emptyset$, i.e., does the SOLP collection $\mathcal{P}_1, \ldots, \mathcal{P}_n$ have any social model?

Theorem 3. The problem SOS_n is NP complete.

6 Knowledge Representation with SOLP Programs

In this section, we provide a real-life example showing the capability of our language of representing common knowledge.

Seating. We must arrange a seating for a number n_{agent} of agents, with m tables and a maximum of c chairs per table. Agents who like (resp. dislike) each other should (resp. should not) sit at the same table. Moreover, an agent can express some requirements w.r.t. the number and the identity of other agents sitting at the same table. Assume that the i-th agent is represented by a predicate $agent(i)$ $(1 \leq i \leq n_{agent})$ and his knowledge base is included in a single SOLP program. The predicate $like(i)$ (resp. $dislike(i)$) means that $Agent_i$ is desired (resp. not tolerated) at the same table, $table(T)$ represents a table $(1 \leq T \leq m)$ and $at(T)$ expresses the desire to sit at table T. For instance, the program $\mathcal{P}_1$ (which is associated to $Agent_1$) could be written as follows:

$$
\begin{array}{rrl}
r_1: & agent(1) & \leftarrow \\
r_2: & & \leftarrow at(T1), at(T2), T1 <> T2 \\
r_3: & at(T) & \leftarrow [, c-1]\{at(T), agent(P)\}, like(P), table(T) \\
r_4: & & \leftarrow at(T), [1,]\{at(T), agent(P)\}, dislike(P) \\
r_5: & & \leftarrow like(P), dislike(P) \\
r_6: & like(2) & \leftarrow \\
r_7: & dislike(3) & \leftarrow \\
r_8: & okay(like(4)) & \leftarrow \\
r_9: & & \leftarrow at(T), [3,]\{at(T)\}
\end{array}
$$

where rules from r_1 to r_5 are common to all the programs (of course, the argument of $agent()$ in r_1 is suited to the enclosing program) and rules from r_6 to r_9 express agent's own requirements. In particular, while the rule r_2 states that any agent cannot be seated at more than one table, the rules r_3 and r_4 mean that an agent wants to share the table with no more than $c-1$ agents he likes and with no agent he dislikes, respectively. The rule r_5 provides consistency for *like* and *dislike*. The rule r_8 is used to declare that $Agent_1$ tolerates $Agent_4$, i.e. $Agent_4$ possibly shares a table with $Agent_1$, and finally the rule r_9 means that $Agent_1$ does not want to share a table with 3 agents or more.

Observe that while the rule r_3 generates possible seating arrangements, the rules r_2, r_4 and r_9 discard those which are not allowed.

7 Conclusions

In this work we have proposed a new language, *SOcial Logic Programming* (SOLP), which enables social behavior among a community of agents represented by logic programs, extending COLP [5]. Thus, the intended models of the *Social Semantics* represent those mental states satisfying social requirements imposed by the agents. Moreover, we have given a translation from SOLP to logic programming with aggregates and discussed the computational complexity of SOLP, which has been proved to be NP complete.

References

1. C. Baral and M. Gelfond. Logic Programming and Knowledge Representation. *Journal of Logic Programming*, 19/20:73–148, 1994.
2. M. Bratman. *Intention, Plans, and Practical Reason.* Harvard Univ. Press, 1987.
3. M. Bratman, D. J. Israel, and M. E. Pollack. Plans and Resource-Bounded Practical Reason. AI center technical note SRI-AI 425R, SRI International, 1988.
4. F Buccafurri and G. Caminiti. A Social-Oriented Semantics for Multi-Agent Systems. Technical Report - TR Lab. Ing. Inf. 05/01, DIMET - University of Reggio Calabria, 2005. Available from the authors.
5. F. Buccafurri and G. Gottlob. *Multiagent Compromises, Joint Fixpoints, and Stable Models*, volume 2407 of *LNCS and LNAI.* Springer, 2002.
6. P. R. Cohen and H. Levesque. Rational interaction as the basis for communication. In *Intentions in Communication.* MIT Press, 1990.
7. R. S. Cost, T. Finin, and Y. Labrou. Coordinating Agents Using ACL Conversations. In *Coordination of Internet Agents: Models, Technologies, and Applications*, pages 183–196, 2001.
8. T. Dell'Armi, W. Faber, G. Ielpa, N. Leone, and G. Pfeifer. Aggregate Functions in Disjunctive Logic Programming: Semantics, Complexity, and Implementation in DLV. In *IJCAI-03, Proc. of the 18th Int. Joint Conf. on Artificial Intelligence*, pages 847–852, 2003.
9. T. Eiter, W. Faber, N. Leone, and G. Pfeifer. Declarative problem-solving in DLV. In *Logic-Based Artificial Intelligence*, pages 79–103. Kluwer, 2000.
10. M. Gelfond and V. Lifschitz. The Stable Model Semantics for Logic Programming. In *5th Conf. on Logic Programming*, pages 1070–1080. MIT Press, 1988.
11. V. Mascardi, M. Martelli, and L. Sterling. Logic-based specification languages for intelligent software agents. *TPLP*, 4(4):429–494, 2004.
12. J. Minker and D. Seipel. *Disjunctive Logic Programming: A Survey and Assessment*, volume 2407 of *LNCS and LNAI.* Springer, 2002.
13. R. S. Patil, R. E. Fikes, P. F. Patel-Schneider, D. McKay, T. Finin, T. Gruber, and R. Neches. The DARPA knowledge sharing effort: progress report. In *Principles of KR and Reasoning: Proc. of the 3rd Int. Conf.* Kaufmann, 1992.
14. W. van der Hoek and W. Wooldrige. Towards a logic of rational agency. *Logic Journal of the IGPL*, 11(2):135–159, 2003.
15. M. Wooldridge. *Reasoning about Rational Agents.* Intelligent Robots and Autonomous Agents. MIT Press, Cambridge, Massachusetts, 2000.
16. M. Wooldridge and N. R. Jennings. Intelligent Agents: Theory and Practice. *The Knowledge Engineering Review*, 2(10):115–152, 1995.

Revisiting the Semantics of Interval Probabilistic Logic Programs

Alex Dekhtyar[1,⋆] and Michael I. Dekhtyar[2,⋆⋆]

[1] Department of Computer Science, University of Kentucky
dekhtyar@cs.uky.edu
[2] Department of Computer Science, Tver State University
Michael.Dekhtyar@tversu.ru

Abstract. Two approaches to logic programming with probabilities emerged over time: bayesian reasoning and probabilistic satisfiability (PSAT). The attractiveness of the former is in tying the logic programming research to the body of work on Bayes networks. The second approach ties computationally reasoning about probabilities with linear programming, and allows for natural expression of imprecision in probabilities via the use of intervals.

In this paper we construct precise semantics for one PSAT-based formalism for reasoning with inteval probabilities, probabilistic logic programs (p-programs), orignally considered by Ng and Subrahmanian. We show that the probability ranges of atoms and formulas in p-programs cannot be expressed as single intervals. We construct the prescise description of the set of models of p-programs and study the computational complexity if this problem, as well as the problem of consistency of a p-program. We also study the conditions under which our semantics coincides with the single-interval semantics originally proposed by Ng and Subrahmanian for p-programs. Our work sheds light on the complexity of construction of reasoning formalisms for imprecise probabilities and suggests that interval probabilities alone are inadequate to support such reasoning.

1 Introduction

Reasoning with probabilistic information, in the context of logic programming, has two distinct origins: bayesian reasoning and probabilistic satisfiability. The former is based on interpreting statements about conditional probability of event A given event B as an implication of a special kind (if B then the probability of A is equal to p). Among the logic programming frameworks following this idea are the work of Poole[16], Ngo and Haddawy [14], and more recently, and in the context of answer set programming, of Baral, Gelfond and Rushton [2].

The second approach to reasoning with probabilistic information starts with Porbabilistic Satisfiability (PSAT), a problem originally formulated by Boole in [1], "resurrected" by Hailperin[9] more than a century later, and, finally, "modernized" by Georgakopoulos, Kavvadis and Papadimitriou[8] in 1988. PSAT is the problem of determining, whether a set $\{P(F) = p_F\}$, of assignments of probabilities to a collection

⋆ Work partially supported by NSF grant ITR-0325063.

⋆⋆ Work partially supported by Russian Foundation for Basic Research grants 04-01-00015 and 04-01-565.

C. Baral et al. (Eds.): LPNMR 2005, LNAI 3662, pp. 330–342, 2005.
© Springer-Verlag Berlin Heidelberg 2005

$\mathcal{F} = \{F\}$ of boolean formulas over atomic events is consistent, i.e., whether there exists a way to assign probabilities to all atomic events in a way that $P(F) = p_F$ for all formulas F in $\mathcal{F}$. Nilsson's probabilistic logic[15] is based on PSAT and uses the semantics of possible worlds (world probability functions) to model probabilities of events. In [8] it is shown that PSAT is NP-complete.

The attractiveness of building logic programming frameworks based on bayesian reasoning lies in direct relationship to the large body of work on Bayesian networks and Markov Decision Processes. The attractiveness of PSAT-based logic programs is in the fact that PSAT has a natural extension to the case of imprecise probabilities. The importance of imprecise probabilities has been observed by numerous researchers in the past 10-15 years [17,3] and lead to the establishment of the Imprecise Probabilities Project [10].

Interval PSAT is a reformulation of PSAT, in which probability assignments of the form $P(F) = p_F$ are relplaced with inequalities of the form $l_F \leq P(F) \leq u_F$. The underlying semantics and the methodology for solving Interval PSAT is the same as for PSAT. Logic programming frameworks inspired by PSAT consider rules of the form "$P(F) = \mu$ if $P(F_1) = \mu_1$ and $\ldots$ and $P(F_n) = \mu_n$". Unlike in bayesian-inspired frameworks, here "if" is the classical logical implication. Logic programming formalisms stemming from PSAT, in which probabilities of events are expressed as intervals, have been considered by Ng and Subrahmanian[11,12] and by Dekhtyar and Subrahmanian[6]. In these frameworks, the fixpoint semantics of formulas, i.e., the set of possible probability assignments for them, had been represented using a single interval.

In [5] we have established that even for simple logic programs (a subset of programs considered by [11]), which contain only atomic events in heads and bodies, the single-interval fixpoint *does not adequately describe* the exact set of possible probability assignments. We have shown that the "real" possible-world semantics is a union of a set of sub-intervals of [0,1].

In this paper, we extend the results of [5] onto the general case of propositional interval probabilistic logic programs as defined in [12]. We formally define the propositional interval probabilistic logic programs of [12][1] in Section 2, where we also show that the single-interval fixpoint is not precise. In Section 3 we provide the precise description of the set of models for an interval logic program. In Section 4 we address the problem of determining if an interval logic program has a model. In Section 5 we study the problem of when the single-interval fixpoint describes all the models of an interval logic program precisely, and prove a number of sufficient conditions.

2 Interval Probabilistic Logic Programs

2.1 Syntax

In this section we describe interval Probabilistic Logic Programs of Ng and Subrahmanian [11,12]. Let L be some first order language containing infinitely many vari-

[1] Ng and Subrahmanian consider in [12] probabilistic logic programs with variables in the probability intervals. In this paper, we consider only constant probability intervals, leaving the rest of the syntax from [12] the same.

able symbols, finitely many predicate symbols and no function symbols. Let $B_L = \{A_1, \ldots, A_N\}$ be the Herbrand base of L. A *basic formula* is either an atom from B_L or a conjunction or disjunction of two or more atoms. The set of all basic formulas is denoted $bf(B_L)$. Formulas of the form $(B_1 \wedge \ldots \wedge B_n) : \mu$ and $(B'_1 \vee \ldots \vee B'_m) : \mu'$, where $B_1, \ldots, B_n, B'_1, \ldots, B'_m \in B_L$ and $\mu = [l, u], \mu' = [l', u'] \subseteq [0, 1]$ are called *p-annotated conjunctions* and *p-annotated disjunctions* respectively.

P-annotated conjunctions and disjunctions represent probabilistic information. Every atom in B_L is assumed to represent an (uncertain) event or statement. A *p-annotated conjunction* $A_1 \wedge \ldots \wedge A_n : [l, u]$ is read as "the probability of the joint occurrence of the events corresponding to $A_1, \ldots, A_n$ lies in the interval $[l, u]$". Similarly, $A_1 \vee \ldots \vee A_n : [l, u]$ is read as "the probability of the occurrence of at least one of the events corresponding to $A_1, \ldots, A_n$ lies in the interval $[l, u]$".

Probabilistic Logic Programs (p-programs) are constructed from *p-annotated* formulas as follows. Let $F, F_1, \ldots, F_n$ be some basic formulas and $\mu, \mu_1, \ldots, \mu_n$ be subintervals of $[0, 1]$ (also called *annotations*). Then, a *p-clause* is an expression of the form $F : \mu \longleftarrow F_1 : \mu_1 \wedge \ldots \wedge F_n : \mu_n$ (if $n = 0$, as usual, the *p-clause* $F : \mu \longleftarrow$ is referred to as a *fact*). A Probabilistic Logic Program (*p-program*) is a finite collection of *p-clauses*. In this paper, we call a p-program in which all clauses consist of atoms from B_L only *a simple p-program*[5]. We also call a p-program in which the heads of all clauses are atoms from B_L a *factored p-program*.

In [11] Ng and Subrahmanian considered factored p-programs. In [13] they considered a framework, in which variables were allowed in the probability annotations. Our definition of p-programs allows arbitrary heads of p-clauses, but does does not consider variable annotations.

2.2 Model Theory

The model theory assumes that in the real world each atom from B_L, and therefore each basic formula, is either true or false. However, exact information about the real world is not known. The uncertainty about the world is represented in a form of a probability distribution over the set of 2^N possible worlds. In addition, p-programs introduce uncertainty about the probability distribution itself.

More formally, given B_L, a world probability density function WP is defined as $WP : 2^{B_L} \rightarrow [0, 1]$, $\sum_{W \subseteq 2^{B_L}} WP(W) = 1$. Each subset W of B_L is considered to be a *possible world* and WP associates a point probability with it. $W \models A$ iff $A \in W$; $W \models A_1 \wedge \ldots \wedge A_n$ iff $(\forall 1 \leq i \leq n) W \models A_i$ and $W \models A_1 \vee \ldots \vee A_n$ iff $(\exists 1 \leq i \leq n) W \models A_i$. We fix an enumeration $W_1, \ldots W_M$, $M = 2^N$ of the possible worlds and denote $WP(W_i)$ as p_i.

Given a function WP, *probabilistic interpretation* (*p-interpretation*) I_{WP} is defined on the set of all basic formulas as follows: $I_{WP} : bf(B_L) \rightarrow [0, 1]$, $I_{WP}(F) = \sum_{W \models F} WP(W)$[2]. P-interpretations assign probabilities to basic formulas by adding up the probabilities of all worlds in which they are true.

[2] Note, that each world probability density function WP has a unique p-interpretation I_{WP} associated with it. However, in general, a p-interpretation I can be induced by more than one world probability density function.

P-interpretations specify the model-theoretic semantics of p-programs. Given a p-interpretation I, the following definitions of satisfaction are given:

– $I \models F : \mu$ **iff** $I(F) \in \mu$;
– $I \models F_1 : \mu_1 \wedge \ldots \wedge F_n : \mu_n$ **iff** $(\forall 1 \leq i \leq n)(I \models F_i : \mu_i)$;
– $I \models F : \mu \longleftarrow F_1 : \mu_1 \wedge \ldots \wedge F_n : \mu_n$ **iff** either $I \models F : \mu$ or $I \not\models F_1 : \mu_1 \wedge \ldots \wedge F_n : \mu_n$.

Now, given a p-program P, $I \models P$ (*I is a model of P*) iff for all p-clauses $C \in P$, $I \models C$. Let $Mod(P)$ denote the set of all models of p-program P. It is convenient to view a single p-interpretation I as a point $(I(F_1), \ldots, I(F_M))$ in $M = 2^N$-dimensional unit cube E^M. Then, $Mod(P)$ can be viewed as a subset of E^M. P is called *consistent* iff $Mod(P) \neq \emptyset$, otherwise P is called *inconsistent*.

2.3 Interval Fixpoint

In this section we give a brief definition of the fixpoint semantics proposed in [12]. The fixpoint semantics of defined on *atomic functions* and *formula functions*.

Let $\mathcal{C}[0,1]$ denote the set of all subintervals of the interval $[0,1]$. An atomic function is a mapping $f : B_L \rightarrow \mathcal{C}[0,1]$. A formula function h is a mapping $h : bf(B_L) \rightarrow \mathcal{C}[0,1]$. Given a set $\mathcal{F} \subseteq bf(B_L)$ a *restricted formula function* is a mapping $f_{\mathcal{F}} : \mathcal{F} \rightarrow \mathcal{C}[0,1]$. Intuitively atomic and formula functions assign probability intervals to atoms and basic formulas: $h(F) = [l,u]$ can be interpreted as the statement ``probability of formula F lies in the interval $[l,u]$".

Each formula function $h_{\mathcal{F}}$ induces a set $\mathcal{LC}(h_{\mathcal{F}})$ of linear inequalities on the probabilities $p_1, \ldots, p_M$ of possible worlds. $\mathcal{LC}(h_{\mathcal{F}})$ consists of the following inequalities:

– $l_F \leq \sum_{W_j \models F} p_j \leq u_F$, for all $F \in \mathcal{F}$, $h_{\mathcal{F}}(F) = [l_F, u_F]$;
– $\sum_{j=1}^{M} p_j = 1$;
– $p_j \geq 0$, for all $1 \leq j \leq M$.

Note that $\sum_{W_j \models F} p_j$ is the probability of F. Therefore, the first group of inequalities specifies, in terms of probabilities p_j of possible worlds, the fact that the probability of F must be between l_F and u_F. The equality $\sum_{j=1}^{M} p_j = 1$ simply states that the sum of probabilities of all possible worlds adds up to 1, while inequalities of the form $p_j \geq 0$ specify that probabilities of possible worlds are nonnegative.

Given a p-program P, two operators, S_P and T_P are defined. They map formula functions to formula functions in the following manner. For a basic formula F, $S_P(h)(F) = \cap M_F$, where $M_F = \{\mu | F : \mu \longleftarrow F_1 : \mu_1 \wedge \ldots \wedge F_n : \mu_n \in P$, and $(\forall 1 \leq i \leq n)(h(F_i) \subseteq \mu_i)\}$. If $M_F = \emptyset$ then $S_P(h)(F) = [0,1]$. The T_P operator is defined as follows: $T_P(h)(F) = [l_F, u_F]$, where $l_F = \min\left(\sum_{W_j \models F} p_j\right)$, subject to $\mathcal{LC}(S_P(h))$ and $u_F = \max\left(\sum_{W_j \models F} p_j\right)$, subject to $\mathcal{LC}(S_P(h))$.

Intuitively, S_P computes the intervals of formulas based on the p-clauses that fired. However, because basic formulas are not, in general, independent (e.g. such formulas as $a \wedge b$ and $a \wedge c$), the ranges computed by S_P may need tightening, performed by T_P. We also note that for factored p-programs, T_P can be specified in a simpler way as described in [11,5]. The work of these operators is illustrated on the following example.

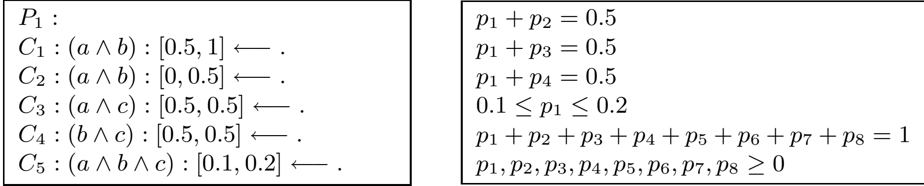


Fig. 1. Sample p-program P_1, and the set of inequalities $\mathcal{LL}(S_P)$ it induces

Example 1. Consider the p-program P_1 shown in Figure 1. Let $h(F) = [0, 1]$ for all $F \in bf_{(}B_L)$. $S_P(h)(a \wedge b) = [0, 0.5] \cap [0.5, 1] = [0.5, 0.5]$. $S_P(h)(a \wedge c) = [0.5, 0.5]$; $S_P(h)(b \wedge c) = [0.5, 0.5]$ and $S_P(h)(a \wedge b \wedge c) = [0.1, 0.2]$. To compute $T_P(h)$ we first construct $\mathcal{LL}(S_P(h)$. Let $W_1 = \{a, b, c\}$, $W_2 = \{a, b\}$, $W_3 = \{a, c\}$ and $W_4 = \{b, c\}$. The set of inequalities $\mathcal{LL}(S_P)(h)$ is shown in Figure 1 (for simplicity replace constraints of the form $a \leq X \leq a$ with $X = a$).

Combining the first three constraints with the fifth we get $2p_1 - 0.5 = p_5 + p_6 + p_7 + p_8$ or $p_1 = 0.25 + p_5 + p_6 + p_7 + p_8$. Because all $p_i \geq 0$, $\min(p_1)$ subject to the latter constraint is 0.25 (when all p_5,p_6,p_7,$p_8 = 0$). However, this contradicts the fourth constraint above which says, in particular $p_1 \leq 0.2$. Thus, $\mathcal{LL}(h)(S_P)$ has no solutions.

Example 2. Consider the p-program $P_2 = P_1 - \{C_5\}$. The computation of $S_P(h)$ will be the same as in the previous example, except $S_P(h)(a \wedge b \wedge c) = [0, 1]$. Now, $T_P(h)(a \wedge b \wedge c)$ is defined: $\min(p_1)$ subject to $\mathcal{LL}(S_P)(h)$ is 0.25 (see previous example for derivation). $\max(p_1) = 0.5$ and it is reached when $p_2 = p_3 = p_4 = 0$. Thus, $T_P(h)(a \wedge b \wedge c) = [0.25, 0.5]$.

The set of all formula functions over $bf(B_L)$ forms a complete lattice $\mathcal{FF}$ w.r.t. the subset inclusion: $h_1 \leq h_2$ iff $(\forall F \in bf(B_L))(h_1(F) \supseteq h_2(F))$. The bottom element $\perp$ of this lattice is the function that assigns $[0, 1]$ interval to all formulas, and the top element $\top$ is the atomic function that assigns $\emptyset$ to all formulas. Ng and Subrahmanian show that T_P is monotonic [11] w.r.t. $\mathcal{FF}$. The iterations of T_P are defined in a standard way: (i) $T_P^0 = \perp$; (ii) $T_P^{\alpha+1} = T_P(T_P^\alpha)$, where $\alpha + 1$ is the successor ordinal whose immediate predecessor is α; (iii) $T_P^\lambda = \sqcup\{T_P^\alpha | \alpha \leq \lambda\}$, where λ is a limit ordinal. Ng and Subrahmanian show that, the least fixpoint $lfp(T_P)$ of the T_P operator is reachable after a finite number of iterations ([11], Theorem 2). They also show that if a p-program P is consistent, then $\mathcal{I}(lfp(T_P))$, the set of all p-interpretations satisfying $lfp(T_P)$[3], contains $Mod(P)$ ([11,12] Corollary 3).

2.4 Fixpoint Is Not Enough

The inverse of the latter statement, however, is not true. We illustrate it on the examples below. There, and elsewhere in the paper, we use the following conventions concerning

[3] $I \models h$ iff for all $F \in bf(B_L)$, $I(F) \in h(F)$ and there exists WP, s.t., WP satisfies $\mathcal{LL}(h)$ and $I = I_{WP}$.

P_3 :	P_4 :
$C_1 : (a \wedge b) : [0.2, 0.5] \longleftarrow .$	$C_1 : (a \wedge b) : [0.2, 0.5] \longleftarrow .$
$C_2 : (c \vee d) : [0.4, 0.6] \longleftarrow .$	$C_2 : (c \vee d) : [0.4, 0.6] \longleftarrow .$
$C_3 : (c \vee d) : [0.5, 0.6] \longleftarrow (a \wedge b) : [0.2, 0.4].$	$C_3 : (c \vee d) : [0.7, 0.8] \longleftarrow (a \wedge b) : [0.2, 0.4].$
$C_4 : (c \vee d) : [0.4, 0.5] \longleftarrow (a \wedge b) : [0.4, 0.5].$	$C_4 : (c \vee d) : [0.7, 0.8] \longleftarrow (a \wedge b) : [0.4, 0.5].$

Fig. 2. Fixpoint does not describe exactly all models for p-programs P_3 and P_4

the possible worlds $W_1, \ldots, W_M$ over which world probability functions are defined. Let $B_L = \{A_1, \ldots, A_N\}$. The mapping of indexes i of worlds W_i to subsets of B_L is the reverse lexicografical order: $W_1 = B_L$, $W_2 = B_L - \{A_N\}, \ldots, W_M = \emptyset$.

Consider now the p-program P_3 in Figure 2.

Proposition 1. *There exists a p-interpretation I, such that $I \models lfp(T_{P_3})$, but $I \not\models P_3$.*

Proof. First, we compute $lfp(T_{P_3})$. On step 1 of the itrative process, $S_{P_3}(\bot)(a \wedge b) = [0.2, 0.5]$ and $S_{P_3}(\bot)(c \vee d) = [0.4, 0.6]$, i.e., clauses C_1 and C_2 of the program will fire. The following constraints are present in $\mathcal{LC}(S_{P_3}(\bot))$.
$0.2 \leq p_1 + p_2 + p_3 + p_4 \leq 0.5$
$0.4 \leq p_1 + p_2 + p_3 + p_5 + p_6 + p_7 + p_9 + p_{10} + p_{11} + p_{13} + p_{14} + p_{15} \leq 0.6$

From these constraints we can find the upper and lower bounds of T_{P_3} on individual atoms. For a we get $l_a = \min(p_1 + p_2 + p_3 + p_4 + p_5 + p_6 + p_7 + p_8) = 0.2$, while $u_a = \max(p_1 + p_2 + p_3 + p_4 + p_5 + p_6 + p_7 + p_8) = 1$. The probability range for b, $[l_b, u_b] = [l_a, u_a] = [0.2, 1]$ due to symmetricity of conjunction of two events. Similarly, we can discover that $l_c = l_d = 0$ and $u_c = u_d = 0.6$.

On the second step, no new rules will fire. Indeed, for the p-clause C_3 to fire, we must have $T^1_{P_3}(a \wedge b) \subseteq [0.2, 0.4]$, and for C_4 to fire, it should be $T^1_{P_3}(a \wedge b) \subseteq [0.4, 0.5]$. But $T^1_{P_3}(a \wedge b) = S_P(\bot)(a \wedge b) = [0.2, 0.5]$, which is a subset of neither $[0.2, 0.4]$ nor $[0.4, 0.5]$. Thus, $lfp(T_{P_3}) = T^1_{P_3}$.

We now show, that there exist a p-interpretation I, such that $I \models T^1_{P_3}$ but $I \not\models P_3$. Consider a (partially defined) p-interpretation I, such that $I(a \wedge b) = 0.3$ and $I(c \vee d) = 0.4$. We complete the construction of I to ensure that it satisfies $T^1_{P_3}$ as follows.
$I(a \wedge b) = p_1 + p_2 + p_3 + p_4 = 0.3$
$I(c \vee d) = p_1 + p_2 + p_3 + p_5 + p_6 + p_7 + p_9 + p_{10} + p_{11} + p_{13} + p_{14} + p_{15} = 0.4$

Let $p_1 = p_2 = p_3 = 0.05$, $p_4 = 0.15$, $p_5 = 0.05$, $p_6 = 0.1$, $p_7 = 0.1$, $p_9 = p_{10} = p_{11} = p_{13} = p_{14} = p_{15} = 0$, $p_{12} = 0.15$, $p_{16} = 0.3$. This assignment satisfies all constraints in $\mathcal{LCS}_{P_3}(\bot)$, which means that $I \models T^1_{P_3}$.

$I \models P_3$ iff $I \models C_1$, $I \models C_2$, $I \models C_3$ and $I \models C_4$. We can see easilty that $I \models C_1$ and $I \models C_2$: $I(a \wedge b) = 0.3 \in [0.2, 0.5]$ and $I(c \vee d) = 0.4 \in [0.4, 0.6]$. However, $I \not\models C_3$. Indeed, $I(a \wedge b) = 0.3 \in [0.2, 0.4]$, i.e., the body of C_3 is **satisfied**, but $I(c \vee d) = 0.4 \notin [0.5, 0.6]$, i.e., the head of C_3 is **not satisfied**. ■

Proposition 1 shows that not all p-interpretations satisfying $lfp(T_P)$ satisfy the program itself, i.e., $Mod(P) \neq lfp(T_P)$. As it turns out, there exist p-programs with non-

empty $lfp(T_P)$ for which $Mod(P) = \emptyset$. One such example is program P_4 shown in Figure 2.

Proposition 2. *$lfp(T_{P_4})$ is not empty, while $Mod(P_4) = \emptyset$.*

Proof. First we show that there are p-interpretations satisfying $lfp(T_{P_4})$. Using reasoning similar to that in the proof of Proposition 1 we see that on the first step of the fixpoint computation process, clauses C_1 and C_2 will fire, giving rise to $T_{P_4}(\bot)(a \wedge b) = S_{P_4}(\bot)(a \wedge b) = [0.2, 0.5]$ and $T_{P_4}(\bot)(c \vee d) = S_{P_4}(\bot)(c \vee d) = [0.4, 0.6]$. On the second step neither C_3 nor C_4 will fire as $T_{P_4}(\bot)(a \wedge b) = [0.2, 0.5] \not\subseteq [0.2, 0.4]$ and $T_{P_4}(\bot)(a \wedge b) = [0.2, 0.5] \not\subseteq [0.4, 0.5]$. This means $lfp_{T_{P_4}} = T^1_{P_4} = T_{P_4}(\bot)$, which is not empty and thus contains satisfying p-interpretations.

Now we show that $Mod(P_4) = \emptyset$. Let $I \models P_4$ be a p-interpretation. $I \models P_4$, means $I \models C_1$, $I \models C_2$, $I \models C_3$ and $I \models C_3$. From $I \models C_1$ we obtain $I(a \wedge b) \in [0.2, 0.5]$. From $I \models C_2$ we obtain $I(c \vee d) \in [0.4, 0.6]$. **Now, we observe that** $I(a \wedge b) \in [0.2, 0.5]$ **implies that either** $I(a \wedge b) \in [0.2, 0.4)$ **or** $I(a \wedge b) \in (0.4, 0.5]$ **or** $I(a \wedge b) = 0.4$**.** Consider each case separately.

If $I(a \wedge b) \in [0.2, 0.4)$, then I satisfies the body of C_3. Therefore, it must be the case that $I(c \vee d) \in [0.7, 0.8]$. However, because $I \models C_2$, $I(c \vee d) \in [0.4, 0.6]$ which leads to a contradiction. Similarly, $I(a \wedge b) \in (0.4, 0.5]$ makes the body of C_4 satisfied, and thus $I(c \vee d)$ must be in $[0.7, 0.8]$ contradicting the fact that $I \models C_2$. Finally, if $I(a \wedge b) = 0.4$ then the bodies of both C_3 and C_4, and hence I must satisfy their (identical) heads, leading again to $I(c \vee d) \in [0.7, 0.8]$, which contradicts $I \models C_2$. Thus, no p-interpretation I can satisfy P_4. ■

Looking at the proofs of both propositions above we see that the reason for the "bad" behavior of $lfp(T_P)$ lies in the computation of the S_P operator, namely, in the determination when p-claues fire. By definition of S_P, a p-clause C fires if current valuation for each basic formula in the body of the clause is a subinterval of its annotation in the clause. Consider, for example a clause $C : F : \mu \longleftarrow G : \mu'$ and some formula function (valuation) h, such that $h(G) \not\subseteq \mu'$ but $h(G) \cap \mu' \neq \emptyset$. This clause will not fire. However, any p-interpretation $I \models C$ such that $I(G) \in h(G) \cap \mu'$, satisfies the body of the clause, and thus, must satisfy its head, i.e., we must have $I(F) \in \mu$. This extra restriction on the probability range of F is not captured by the S_P computation.

3 Possible Worlds Semantics

We ask ourselves: given a p-program P, how do we give an exact description of $Mod(P)$? In [5] we have answered this question of simple p-programs, i.e., p-programs with only atoms in the program clauses. In this section we extend the new semantics to the full case of p-programs.

Definition 1. *Let P be a p-program over the Herbrand base $B_L = \{A_1, \ldots, A_N\}$, and let $\mathcal{W} = (W_1, \ldots, W_M)$, $M = 2^N$ be an enumeration of all subsets of B_L. With each W_j, $1 \leq j \leq M$ we associate a variable p_j with domain $[0, 1]$. Let C be a p-clause in P of the form $F : [l, u] \longleftarrow F_1 : [_1, u_1] \wedge \ldots \wedge F_n : [l_n, u_n]$.*

The family of systems of inequalities induced by C*, denoted $INEQ(C)$ is defined as follows:*

– $n = 0$ *(C is a fact).* $INEQ(C) = \left\{l \leq \sum_{W_j \models F} p_j \leq u\right\}$.
– $n \geq 1$ *(C is a rule).* $INEQ(C) = T(C) \cup F(C)$;
$T(C) = \left\{\left\{l \leq \sum_{W_j \models F} p_j \leq u; l_i \leq \sum_{W_j \models F_i} p_j \leq u_i | 1 \leq i \leq k\right\}\right\}$;
$F(C) = \left\{\left\{\sum_{W_j \models F_i} p_j < l_i\right\} | 1 \leq i \leq k\right\} \cup \left\{\left\{\sum_{W_j \models F_i} p_j > u_i\right\} | 1 \leq i \leq k\right\}$.

Let $P = \{C_1, \ldots, C_s\}$*. Then,* $INEQ(P)$ *is defined as follows:*
$INEQ(P) = \{\alpha_1 \cup \ldots \cup \alpha_s | \alpha_i \in INEQ(C_i), 1 \leq i \leq s\}$

Informally, $INEQ(P)$ is constructed as follows: for each p-clause C in the program we select the reason, why it is true. The reason/evidence is either the statement that the head of the clause is satisfied, or that one of the conjuncts in the body is not. The set $INEQ(P)$ represents all possible systems of such evidence/restrictions on probabilities of basic formulas. Solutions of any system of inequalities in $INEQ(P)$ satisfy every clause of P. Of course, not all individual systems of inequalities have solutions, but $INEQ(P)$ *captures all the systems that do*, as shown in the following lemma and theorem.

Lemma 1. *Let* $C : F : \mu \longleftarrow F_1 : \mu_1 \wedge \ldots \wedge F_m : \mu_k$ *be a p-clause and* I *be a* $p\!-\!interpretation$ *(both over the same Herbrand Base* B_L*). Then* $I \models C$ *iff there exists a world probability function* WP*, such that* $I = I_{WP}$ *and* $\{p_j = WP(W_j) | W_j \subseteq B_L\}$ $\in Sol(\alpha)$ *for some* $\alpha \in INEQ(C)$*.*

Proof (*sketch*). First we note that given a p-interpretation I, one can always construct a world probability function WP, such that $I = I_W P$ [11,12]. If $I \models C$, then either either I satisfies $F : \mu$, the head of C, or it does not satisfy the body of C. In the first case, we show in a straightforward manner that any world density function WP, such that $I = I_{WP}$ is in the solution of $T(C)$. In the second case, we determine which conjunt $F_i : \mu_i$ in the body is not satisfied, and show that the appropriate system of inequalities from $F(C)$ has any such WP as a solution. Going back, if WP is a probability density function that is a solution of some system of inequalities $\alpha \in INEQ(C)$, we show that $I_W P \models C$ as follows. α must be either in $T(C)$ or in $F(C)$. In the first case, we show in a straightforward manner that $I(F) \in \mu$. In the second case, we show that $I(F) \notin \mu_i$ for some $1 \leq i \leq m$. ■

Theorem 1. *A p-interpretation* I *is a model of a p-program* P ***iff*** *there exists a world probability function* WP*, such that* $I = I_{WP}$*, and a system of inequalities* $\alpha \in INEQ(P)$ *such that* $\mathcal{P} = \{p_j = WP(W_j) | W_j \subseteq B_L\} \in Sol(\alpha)$*.*

This leads to the following description of $Mod(P)$:

Corollary 1. $Mod(P) = \bigcup_{\alpha \in INEQ(P)} \{I_{WP} | WP \in Sol(\alpha)\}$

Let $Rules(P)$ and $Facts(P)$ denote the sets of p-clauses from P with non-empty and empty bodies respectively. Let $f(P) = |Facts(P)|$ and $r(P) = |Rules(P)|$. Finally, let $k(P)$ be the maximum number of basic formulas in a body of a rule in P.

The solution of each system $\alpha \in INEQ(P)$ is a convex $M - 1$-dimensional[4] (in general case) polyhedron. Given a solution WP of some $\alpha \in INEQ(P)$, I_{WP} is

[4] Because $p_1 + \ldots + p_M = 1$ is present in every $\alpha \in INEQ(P)$, the dimensionality of $Sol(\alpha)$ cannot be more than $M - 1$.

obtained via a linear transformation. Because linear transformations preserve convexity of regions, we can make the following statement about the geometry of the set $Mod(P)$.

Corollary 2. *Given a p-program P over the Herbrand base $B_L = \{A_1, \dots, A_N\}$, $Mod(P)$ is a union of $S \leq (2k(P)+1)^{r(P)}$, not necessarily disjoint, convex polyhedra. Each polyhedron has a dimensionality of at most $M - 1 = 2^N - 1$.*

This corollary provides an exponental, in the size of the p-program, upper bound on the number of *possibly disjoint* components of $Mod(P)$. In [5] we constructed a simple p-program P with $2N + 1$ clauses and $k(P) = 1$, whose $Mod(P)$ is a collection of 2^N disjoint N-dimensional parallepipeds. This shows that the exponential bound cannot be substantially decreased.

The semantics of p-programs is closely connected to Interval PSAT. As mentioned above, each system of inequalities in $INEQ(P)$ is constructed by selecting one formula from each clause (either the head or from the body) and assigning it an interval: $[l, u]$ for the head; $[0, l_i)$ or $(u_i, 1]$ for the formula F_i from the body. Theorem 1 showed that any assignment of point probabilities to the atoms, that satisfies these constraints is a model of P. At the same time, the set $\{F : \mu\}$ of annotated formulas for which satisfying p-interpretations are to be found is an *instance of Interval PSAT*. Thus, an instance of Interval PSAT is associated with each set of inequalities in $INEQ(P)$. We note that the sets of solutions for individual systems from $INEQ(P)$ are not disjoint, however, each system can contain unique solutions. Thus, one way of computing $Mod(P)$ is to solve $|INEQ(P)|$ Interval PSAT problems.

4 Consistency Problem

The consistency problem for p-programs is defined as follows: given a p-program P, check whether P has a model, i. e. $Mod(P) \neq \emptyset$. Let CONS-P= $\{P | Mod(P) \neq \emptyset\}$.

Theorem 2. *The set CONS-P is NP-complete.*

Proof. *Upper bound.* Let P be a p-program, $B_1, \dots, B_r$ be all basic formulas of P Then $Mod(P) \neq \emptyset$ iff there exist such probabilitues $b_1, \dots, b_r$ of $B_1, \dots, B_r$ that
(i) the system of linear equations and inequalities $EQ(P)$:

- $\sum_{W_j \models B_i} p_j = b_i$, for $i = 1, \dots, r$,
- $\sum_{j=1}^{M} p_j = 1$;
- $p_j \geq 0$, for all $1 \leq j \leq M$.

has a solution $WP = \{p'_1 \dots, p'_M\}$ defined the interpretation $I_{WP} \in Mod(P)$.

To prove the upper bound, we use the following lemma from [7] (which, in turn, cites [4]. Similar statement is also found in [8]).

Lemma 2. *If a system of r linear equations and/or inequalities with integer coefficients each of length at most l has a nonnegative solution, then it has a nonnegative solution with at most r entries positive, and where the size of each member of the solution is $O(rl + r \log r)$.*

Based on this lemma we obtain the following “small model” theorem.

Lemma 3. *p-program P including r different basic formulas is consistent iff there exists a probability distribution WP on possible worlds with no more than $r+1$ nonzero probabilities such that $I_{WP} \models P$.*

Let the longest number in annotations of P have length l. Then the following nondeterministic procedure allows us to check whether $Mod(P) \neq \emptyset$.

1) Guess for each $B_i (i = 1, \ldots, r)$ it's probability $b_i \in [0, 1]$ of the length $O(rl + r \log r)$.
2) Guess a probability distribution WP with no more than $r+1$ positive probabilities $p_{i_1}, \ldots, p_{i_{r+1}}$ of the length $O(rl + r \log r)$ and check that WP is a solution of the system $EQ(P)$.
3) If $I_{WP} \models P$ retur n "Yes".

From the lemmas above it follows that this algorithm runs in nondeterministic time bounded by a polynomial of $|P|$.

Lower bound. We prove the lower bound for a subclass of simple p-programs with clause bodies of size 3 or less. We show that 3-CNF $\leq_P$ CONS-P. Let $\Phi = C_1 \wedge \ldots C_m$ be a 3-CNF over the set of boolean variables $Var = \{x_1, \ldots, x_n\}$. Let each clause $C_j, j = 1, \ldots, m$, include 3 literals l_j^1, l_j^2, l_j^3. Define for each literal l an annotated atom $\alpha(l)$ as follows: if $l = x \in Var$ then $\alpha(l) = x : [0.5, 1]$, if $l = \neg x$ then $\alpha(l) = x : [0, 0.5]$. Let $B_L = Var \cup \{C_j | \ j = 1, \ldots, m\} \cup \{\Phi\}$. We include in p-program $P(\Phi)$ the following p-clauses. $(f1): \ \Phi : [1.1] \leftarrow .$
$(fc_j): \ C_j : [0, 0.1] \leftarrow . \ (j = 1, \ldots, m)$
$(fx_i): \ x_i : [0, 1] \leftarrow . \ (i = 1, \ldots, n)$
$(rc_j): \ C_j : [0.9, 1] \leftarrow \alpha(l_j^1) \wedge \alpha(l_j^2) \wedge \alpha(l_j^3). \ (j = 1, \ldots, m).$
$(rf_i): \ \Phi : [0, 0] \leftarrow x_i : [0.5, 0.5]. \ (i = 1, \ldots, n)$

It is easy to see that $P(\Phi)$ can be constructed from Φ in polynomial time. Now the theorem follows from the following proposition.

Proposition 3. *$\Phi \in$ 3-CNF $\iff$ $P(\Phi) \in$ CONS-P.*

A consistent p-program P *entails* a formula $F : [l, u]$ if for each $I \in Mod(P)$ $I \models F : [l, u]$. The entailment problem is, thus, expressed as follows: given a consistent P and a formula $F : [l, u]$, decide if P entails $F : [l, u]$?

Let $EQ_1(P, F) = EQ(P) \cup \{\sum_{W_j \models F} p_j < l\}$ and $EQ_2(P, F) = EQ(P) \cup \{\sum_{W_j \models F} p_j > u\}$. Then it easy to see that *P does not entail $F : [l, u]$ iff $EQ_1(P, F)$ is solvable or $EQ_2(P, F)$ is solvable.* Therefore we get the following complexity bounds for the entailment problem.

Theorem 3. *The enailment problem for p-programs is co-NP-complete.*

We note, in fact, that the theorem holds for the class of simple p-programs.

5 When Fixpoint Is Enough?

In this section we study subclasses of p-programs for which simpler procedures for determining $Mod(P)$ exist. In particular, we ask ourselves a question of when $Mod(P)$,

as defined here, and $lfp(T_P)$, as defined in [12] coincide. We then address the problem of complexity of detecting that $Mod(P) = \mathcal{I}(lfp(T_P))$. First, we consider the problem of $Mod(P) = lfp(T_P)$ for the case of simple p-programs. In the definition below we use the following notation. Given a simple p-program P and an atom A, we let $ha_P(A)$ denote the set of all intervals associated with occurrences of a in the heads of clauses in P. We also let $ba_P(A)$ denote the set of all intervals associated with occurrences of A in the bodies of the clauses from P.

Definition 2. *A simple p-program P is called* semi-strict *if it satifies the following condition: for all atoms $A \in B_L$, and for each pair $\mu \in ha_P(A)$ and $\nu \in ba_P(A)$ either $\mu \subseteq \nu$ or $\mu \cap \nu = \emptyset$.*

Intuitively, a simple p-program is called semi-strict if for all atoms their annotations in the heads of the rules are either subintervals of annotations in the bodies or do not intersect with them.

Theorem 4. *If P is a simple semi-strict p-program, then $Mod(P) = \mathcal{I}(lfp(T_P))$.*

Theorem 5. *Semi-strictness of a simple p-program P can be checked in time $O(|P|^2)$.*

Semi-strictness is a syntactic condition on simple p-programs, that can be checked in time, quadratic, in the size of the p-program in a straightforward manner. This makes it an attractive condition to use in general case. However, two facts make it impossible. First, this is a sufficient, but not necessary condition, and second, for programs with non-atomic formulas, semi-strictness does not imply $Mod(P) = \mathcal{I}(lfp(T_P))$. The following two examples illustrate this.

Example 3. To show that semi-strictness is not a necessary condition, consider p-program P_5 from Figure 3. First, we note that $lfp(T_{P_5})$ assigns intervals $[0.2, 0.4]$, $[0.3, 0.7]$ and $[0, 1]$ to atoms a, b and c respectively. We can also see that the body of the third clause of P_5 is unsatisfiable given the first two clauses, because the intervals for a, $[0.5, 1]$ in the clause and $[0.2, 0.4]$ from the first clause, do not intersect. Therefore, $Mod(P_5)$ will not differ from $lfp(T_{P_5})$. At the same time, we note that P_5 is not semi-strict, because for b the annotation of the head of the second clause, $[0.3, 0.7]$, and the annotation in the body of the third clause, $[0.6, 0.9]$ overlap.

Example 4. Consider the p-program P_6 from Figure 3. P_6 is semi-strict by definition 2. But we can show that $\mathcal{I}(lfp(T_P))$ and $Mod(P)$ differ. Indeed, because the constraints on the probabilities of a and b from the first two p-clauses do not entail the $[0.3, 0.6]$

P_5 :	P_6 :
$a : [0.2, 0.4] \longleftarrow .$	$a : [0.2, 0.6] \longleftarrow .$
$b : [0.3, 0.7] \longleftarrow .$	$b : [0.3, 0.7] \longleftarrow .$
$c : [0.8, 0.9] \longleftarrow b : [0.6, 0.9], a : [0.5, 1].$	$c : [0.8, 0.9] \longleftarrow (a \wedge b) : [0.3, 0.6].$

Fig. 3. Programs P_5 and P_6 show that semi-strictness is not the right condition for general p-programs

constraint on the probability of $a \wedge b$, this rule does not fire, and therefore $lfp(T_P)(c) = [0,1]$. In particular, a p-interpretation I, s.t., $I(a) = 0.6$, $I(b) = 0.7$, $I(c) = 0.2$ is in $\mathcal{I}(lfp(T_P))$. At the same time, if $I(a) = 0.6$ and $I(b) = 0.7$, then $I(a \wedge b) \in [0.3, 0.6]$, and therefore, in the thrid rule, the head must be satisfied, but $0.2 \notin [0.8, 0.9]$.

It turns out that it is possible to specify a sufficient condition in the general case. However, this is no longer a syntactic condition.

Definition 3. *Let P be a p-program and let P' be the result of removing from P all p-clauses whose heads are satisfied by $lfp(T_P)$. A p-program P is called* strict *if the following condition holds: for each clause $C : F : \mu \longleftarrow F_1 : \mu_1 \wedge \ldots F_n : \mu_n$ in P', there exists an index $1 \leq i \leq n$, such that $lfp(T_P)(F_i) \cap \mu_i = \emptyset$.*

Theorem 6. *If a p-program P is strict, then $Mod(P) = \mathcal{I}(lfp(T_P))$.*

Proof. We know that $Mod(P) \subseteq \mathcal{I}(lfp(T_P))$. Suppose now, $I \in \mathcal{I}(lfp(T_P))$. We show that $(\forall C : F : \mu \longleftarrow F_1 : \mu_1 \wedge \ldots F_n : \mu_n \in P) I \models C$. If $C \in P - P'$, then $I(F) \in lfp(T_P)(F) \subseteq \mu$, and therefore, $I \models F : \mu$. If $C \in P'$, then, because C is strict, there exists such index i, that $lfp(T_P)(F_i) \cap \mu_i = \emptyset$. Then $I \not\models F_i : \mu_i$, and therefore $I \not\models F : \mu \longleftarrow F_1 : \mu_1 \wedge \ldots F_n : \mu_n$ and $I \models C$. ■

For the class of simple p-programs, strictness can be efficiently checked and is a necessary condition. This leads to polynomial-time upper bounds on entailment and consistency.

Theorem 7. 1. *For a simple p-program P checking whether it is strict can be performed in polynomial time.*
2. *For a simple p-program P, $Mod(P) = \mathcal{I}(lfp(T_P))$ **iff** P is strict.*

Corollary 3. *Consistency and entailment problems are solvable in polynomial time for strict simple p-programs.*

The following example shows that strictness is not a necessary condition for non-simple programs.

Example 5. Consider the following p-program P_7:

$a : [0.6, 0.8] \longleftarrow .$ $\quad b : [0.6, 0.7] \longleftarrow .$ $\quad d : [0.2, 0.3] \longleftarrow .$
$c : [0.4, 0.5] \longleftarrow (a \wedge b) : [0.65, 0.7] \wedge (b \vee d) : [0.5, 0.6].$

$lfp(T_P)$ assigns intervals $[0.2, 0.7]$ and $[0.6, 1]$ to $a \wedge b$ and $b \vee d$ respectively, and therefore, P_7 is *not strict*. However, there exists no p-interpretation I which satisfies the first three rules and the body of the fourth rule: $I(b \vee d) \in [0.5, 0.6]$ implies, $I(b \vee d) = 0.6$ and $I(b) = 0.6$, while $I(a \wedge b) \in [0.65, 0.7]$ implies that $I(b) \geq 0.65$. Therefore, $Mod(P)$ coincides with $\mathcal{I}(lfp(T_P))$.

6 Related Work and Conclusions

A survey of different approaches to probabilistic logic programming can be found in [6] and [5]. This paper studies the precise semantics of a logic programming language

for reasoning about the interval probabilities of events and their combinations. This language, proposed by Ng and Subrahmanian[12] is a natural extension of Interval Probabilistic Satisfiability problem PSAT [8]: an instance of Interval PSAT is a p-program, in which all rules have no bodies. We show that for this, relatively simple language, the class of satisfying models (probabilistic interpretations) has a complex description: it is a union of a number of (closed, open, semiopen) intervals, obtained, solving an array of Interval PSAT problems. On the positive side, our results show how to compute the set of models of a p-program *precisely*. On the negative side, the complexity of the description and the computational complexity of the problem itself suggest that intervals may be inadequate as the means for specifying imprecision in probabilistic assessments.

References

1. G. Boole. (1854) *The Laws of Thought*, Macmillan, London.
2. Chitta Baral, Michael Gelfond, J. Nelson Rushton. (2004) Probabilistic Reasoning With Answer Sets, in *Proc. LPNMR-2004*, pp. 21-33.
3. Luis M. de Campos, Juan F. Huete, Serafin Moral (1994). Probability Intervals: A Tool for Uncertain Reasoning, *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems (IJUFKS)*, Vol. 2(2), pp. 167 – 196.
4. V.Chvátal. (1983) Linear Programming. *W. Freeman and Co.*, San Fancisco, CA.
5. A. Dekhtyar, M.I. Dekhtyar. (2004) Possible Worlds Semantics for Probabilistic Logic Programs, in *Proc., International Conference on Logic Programming (ICLP)'2004, LNCS*, Vol. 3132, pp. 137-148.
6. A. Dekhtyar and V.S. Subrahmanian (2000) Hybrid Probabilistic Programs. *Journal of Logic Programming*, Volume 43, Issue 3, pp. 187 – 250 .
7. R. Fagin J. Halpern, and N. Megiddo. (1990) A logic for reasoning about probabilities, *Information and Computation*, vol. 87, no. 1,2, pp. 78-128.
8. G.Georgakopoulos, D. Kavvadias, C.H. Papadimitriou. (1988) Probabilistic Satisfiability, *Journal of Complexity*, Vol. 4, pp. 1-11.
9. T. Hailperin. (1965) Best Possible Inequalities for the Probability of a Logical Function of Events, *American Mathematical Monthly*, Vol. 72, pp. 343–359.
10. H.E. Kyburg Jr. (1998) Interval-valued Probabilities, in *G. de Cooman, P. Walley and F.G. Cozman (Eds.), Imprecise Probabilities Project*, http://ippserv.rug.ac.be/documentation/interval_prob/interval_prob.html.
11. R. Ng and V.S. Subrahmanian. (1993) Probabilistic Logic Programming, *Information and Computation*, 101, 2, pps 150–201, 1993.
12. R. Ng and V.S. Subrahmanian. A Semantical Framework for Supporting Subjective and Conditional Probabilities in Deductive Databases, JOURNAL OF AUTOMATED REASONING, 10, 2, pps 191–235, 1993.
13. R. Ng and V.S. Subrahmanian. (1995) *Stable Semantics for Probabilistic Deductive Databases*, INFORMATION AND COMPUTATION, 110, 1, pps 42-83.
14. L. Ngo, P. Haddawy (1995) Probabilistic Logic Programming and Bayesian Networks, in *Proc. ASIAN-1995*, pp. 286-300.
15. N. Nilsson. (1986) *Probabilistic Logic*, AI Journal 28, pp 71–87.
16. D. Poole (1993). Probabilistic Horn Abduction and Bayesian Networks. *Artificial Intelligence*, Vol. 64(1), pp. 81-129.
17. Walley, P. (1991). Statistical Reasoning with Imprecise Probabilities. Chapman and Hall, 1991.

Routley Semantics for Answer Sets

Sergei Odintsov[1] and David Pearce[2,⋆]

[1] Sobolev Institute of Mathematics, Novosobirsk, Russia
odintsov@math.nsc.ru
[2] Universidad Rey Juan Carlos, Madrid, Spain
d.pearce@escet.urjc.es

Abstract. We present an alternative model theory for answer sets based on the possible worlds semantics proposed by Routley (1974) as a framework for the propositional logics of Fitch and Nelson. By introducing a falsity constant or second negation into Routley models, we show how paraconsistent as well as ordinary answer sets can be represented via a simple minimality condition on models. This means we can define a paraconsistent version of equilibrium logic, or paraconsistent answer sets (PAS) for propositional theories. The underlying logic of PAS is denoted by $\mathbf{N}_9$. We characterise it axiomatically and algebraically, showing it to be the least conservative extension of the logic of here-and-there with strong negation. In addition, we show that $\mathbf{N}_9$ captures the strong equivalence of programs in the paraconsistent case and can thus serve as a useful mathematical foundation for PAS. We end by showing that $\mathbf{N}_9$ has the Interpolation Property.

1 Introduction

In Pearce [19] it was shown how the nonclassical logic of here-and-there with strong negation, often denoted by $\mathbf{N}_5$, can serve as a foundation for answer set programming (ASP). The main property involved is that answer sets can be viewed as a certain kind of minimal $\mathbf{N}_5$-model. A second key property was established in [13]: programs are strongly equivalent wrt answer set semantics (see §5.2) if and only if they are equivalent viewed as propositional theories in $\mathbf{N}_5$. This shows that $\mathbf{N}_5$ can be used to reason about answer set programs, and $\mathbf{N}_5$-deduction may be relevant for program transformation and optimisation. These issues have been explored in several recent works in the area of logic programming and nonmonotonic reasoning, where $\mathbf{N}_5$-inference as well as its metatheoretic properties have been exploited, [21,26,27].

It is natural to ask whether other kinds of logic programming semantics, either variants of ASP, or alternatives such as the well-founded semantics (WFS), also possess a well-behaved monotonic 'base' logic. Although some partial results have been obtained, generally speaking complete solutions are still lacking. For WFS, for instance, it is not even known whether there exists a monotonic, deductive characterisation of strongly equivalent programs.

In this paper we show how the paraconsistent version of answer set semantics also admits a natural underlying, monotonic logic, which we denote by $\mathbf{N}_9$, and we look at an alternative model theory for answer sets due to R. Routley. Paraconsistent answer

⋆ Partially supported by CICyT project TIC-2003-9001-C02 and WASP (IST-2001-37004).

C. Baral et al. (Eds.): LPNMR 2005, LNAI 3662, pp. 343–355, 2005.
© Springer-Verlag Berlin Heidelberg 2005

sets (PAS) were studied as a logic programming semantics by Sakama and Inoue [30]. Recently, Alcantara *et al* [1] have made some progress towards a logical, declarative style of characterisation for PAS. However, [1] do not axiomatise or otherwise syntactically characterise the underlying (monotonic) logic of PAS; nor do they investigate the problem of strong equivalence. Moreover, their semantical frames can easily be reduced to a simpler notion of possible worlds model proposed by Routley [29] in 1974 as a semantical framework for the propositional logics of Nelson [16] and Fitch [6]. By introducing a falsity constant or second negation into Routley models, we can show how paraconsistent as well as ordinary answer sets can be represented via a simple minimality condition on models. It is relatively straightforward to prove this property from first principles. However, in the paper we will make use of the results of [1]. In particular we show (i) how the semantical frames of [1] can be reduced to Routley models, and (ii) how this leads to a simple characterisation of paraconsistent answer sets. An additional condition on Routley models yields $\mathbf{N}_5$-models that capture ordinary answer sets.

In the case of PAS, the underlying logic $\mathbf{N}_9$ belongs to the lattice of logics studied by Odintsov [17] which we denote here by $\mathcal{E}\mathbf{N}_{\neg}^{-}$. We axiomatise the logic, showing it to be the least conservative extension of the logic of here-and-there with strong negation, representable via the full twist-structure on the 3-element Heyting algebra. The latter means that $\mathbf{N}_9$ can be viewed as a 9-valued logic, which explains the notation. In addition, we show that the logic suffices to characterise the strong equivalence of programs in the paraconsistent case and can thus serve as a useful mathematical foundation for PAS. We conclude the mathematical part of the paper by showing that the logic has the Interpolation Property.

Besides providing a logico-mathematical foundation for PAS, and an extension to arbitrary theories, we hope to shed some light on the interplay between strong or explicit negation, '$\sim$', and default negation or negation-as-failure, '$\neg$' or '*not*'. Unlike in the usual case, in the paraconsistent version of answer sets, default negation is no longer definable in terms of strong negation. An interesting feature of PAS is that all the paraconsistency resides in that part of reasoning involving strong negation, reflecting the idea that basic information (facts) and even rules may be contradictory. That part of reasoning involving default negation, on the other hand, remains quite standard: the underlying logic of PAS extends intuitionistic logic and is even a conservative extension of the logic of here-and-there, when '*not*' is identified with intuitionistic negation '$\neg$'.

2 Some Background on Constructible Falsity

The concept of constructible falsity was introduced into logic by David Nelson [16] via his system of constructive logic with strong negation, later often denoted by $\mathbf{N}$. Nelson's logic was subsequently axiomatised by Vorob'ev [32,33] and studied algebraically in the 1950s and 60s by the group of Helena Rasiowa [28]. A logical system related to Nelson's, but somewhat weaker, was developed by Fitch [6].

From the early 1970s several authors explored logical systems similar to Nelson's and Fitch's but lacking the "explosive" axiom $\alpha \to (\sim\alpha \to \beta)$, thus producing *paraconsistent* logics. The paraconsistent version of $\mathbf{N}$ is sometimes denoted by $\mathbf{N}^{-}$. It was studied independently by R. Routley (later R. Sylvan) in the propositional case in [29],

by López-Escobar in [14] and by Nelson himself in [2], both in the first-order case. A very similar system was explored by von Kutschera in [12]. Both the logics $\mathbf{N}$ and $\mathbf{N}^-$, as well as the corresponding systems of Fitch, are extensions of positive logic. Only $\mathbf{N}$ itself can be viewed as an extension of intuitionistic logic, when the intuitionistic negation '$\neg$' is defined in it, say by $\neg\alpha := \alpha \rightarrow \sim\alpha$.

Kripke semantics for $\mathbf{N}$ ([31,9]) is readily obtained from the usual Kripke semantics for intuitionistic logic by assigning to each world, instead of a set of atoms, a set of *literals*, ie. atoms or strongly negated atoms. Equivalently, the truth-assignment on atoms and worlds is 3-valued, to reflect the three cases of verified, falsified or neither. Changing to 4-valued assignments in Kripke models produces a semantics for $\mathbf{N}^-$: the fourth value now corresponds to "overdetermined" or the situation that both a literal and its contrary are verified at a world. An alternative possible worlds semantics for constructive logic with strong negation was provided by Routley in [29]. Routley studied all four variants of the Fitch and Nelson logics and proved completeness relative to models with 2-valued truth assignments. To handle strong negation and paraconsistency, Routley made use of a kind of non-normal worlds, related to ordinary worlds via a star operation (a kind of involution); hence one has normal and "starred" worlds.

Since in $\mathbf{N}^-$, unlike in $\mathbf{N}$, intuitionistic negation is not definable, there are several options for introducing a second negation into the paraconsistent systems. This topic has recently been studied in [17] which investigates the lattice of extensions of $\mathbf{N}^-$ augmented with a falsum constant $\perp$ satisfying two new axioms.

2.1 Constructible Falsity and Nonmonotonic Reasoning

At the end of the 80s the use of constructible falsity, via normal and paraconsistent versions of Nelson's logic and its variants, was explored as a tool for knowledge representation and reasoning. and studied in depth in the context of logical languages for information and knowledge exchange in several articles and books, [24,25,36,10,35]. In nonmonotonic logic programming in the tradition of stable models, strong negation became firmly established via the answer set semantics of Gelfond and Lifschitz [7]. The fact that they called their second negation, representing explicit falsity, *classical*, obscured at first the connection to Nelson's logic $\mathbf{N}$; though for computational purposes they used precisely the same reduction technique as Vorob'ev to eliminate strong negation through the addition of new predicates. That answer set inference could be viewed as an extension of Nelson's logic $\mathbf{N}$ was later established in [18].

Later, attempts to find a precise match between answer set inference and a logic in the lattice of extensions of $\mathbf{N}$ led to *equilibrium* logic based on the propositional logic of here-and-there with strong negation, see Pearce [19,20], that we denote here by $\mathbf{N}_5$. Equilibrium logic can be understood both as a nonmonotonic extension of $\mathbf{N}_5$ and as a generalisation of the language of answer set programming to full propositional logic.

Paraconsistency also came to be considered in logic programming, initially by Blair and Subrahmanian [3], later by several other authors, for an overview see [4]. Among them, Sakama and Inoue [30] proposed a paraconsistent version of answer set semantics, essentially by dropping the requirement that one discard inconsistent models in the construction of answer sets. Recently the paraconsistent version of answer sets (PAS) was taken up again by Alcantara, Damásio and Pereira [1] who have attempted to give

a more declarative style of characterisation of PAS, more in the spirit of the logical approach to AS [19]. However [1] uses a two-valued semantics based on frames, rather than the usual four values associated with the more typical Kripke model approach to paraconsistent logics such as $\mathbf{N}^-$. Here we show that the Routley [29] semantics, with the addition of a constant $\bot$ or a second negation $\neg$, will do equally well; the advantage is that we know these models to be complete for $\mathbf{N}^-$.

3 Routley Semantics for N^-

$\mathbf{N}^-$ is the weak, paraconsistent version of Nelson's constructive logic with strong negation. Formulas of $\mathbf{N}^-$ are built-up in the usual way using the logical constants: $\wedge$, $\vee$, $\rightarrow$, $\sim$, standing respectively for conjunction, disjunction, implication and strong negation. The only rule of inference for $\mathbf{N}^-$ is *modus ponens* and the axioms are the axiom schemata of positive logic:

P1. $\alpha \rightarrow (\beta \rightarrow \alpha)$
P2. $(\alpha \wedge \beta) \rightarrow \alpha$
P3. $(\alpha \rightarrow (\beta \rightarrow \gamma)) \rightarrow ((\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \gamma))$
P4. $(\alpha \wedge \beta) \rightarrow \beta$
P5. $(\alpha \rightarrow \beta) \rightarrow ((\alpha \rightarrow \gamma) \rightarrow (\alpha \rightarrow (\beta \wedge \gamma)))$
P6. $\alpha \rightarrow (\alpha \vee \beta)$
P7. $(\alpha \rightarrow \gamma) \rightarrow ((\beta \rightarrow \gamma) \rightarrow ((\alpha \vee \beta) \rightarrow \gamma))$
P8. $\beta \rightarrow (\alpha \vee \beta)$

plus the following axiom schemata involving strong negation taken from the calculus of Vorob'ev [32,33] (where '$\alpha \leftrightarrow \beta$' abbreviates $(\alpha \rightarrow \beta) \wedge (\beta \rightarrow \alpha)$):

N1. $\sim (\alpha \rightarrow \beta) \leftrightarrow \alpha \wedge \sim\beta$
N2. $\sim(\alpha \wedge \beta) \leftrightarrow \sim\alpha \vee \sim \beta$
N3. $\sim(\alpha \vee \beta) \leftrightarrow \sim\alpha \wedge \sim\beta$
N4. $\sim \sim\alpha \leftrightarrow \alpha$

The main idea of Routley semantics is that the validity of negation $\sim\alpha$ at a world w is equivalent to the falsity of α not at w as in classical logic but at some adjacent world w^{*}[1]. To define Routley semantics for $\mathbf{N}^-$ we have additionally to divide the set of possible worlds into parts, unstarred and starred worlds. A Routley model for $\mathbf{N}^-$ is a quadruple $\mathcal{M} = \langle W \cup W^*, \leq, ^*, V \rangle$ such that: $W \cup W^*$ is a non-empty set (of worlds), $W \cap W^* = \emptyset$, $\leq$ is a partial ordering on W, * is a bijection on $W \cup W^*$, $*(W) = W^*$, and V is a valuation function from $\mathit{Atoms} \times W \longrightarrow \{0,1\}$ satisfying the following conditions: 1. $u \leq w \Rightarrow w^* \leq u^*$, 2. $w = w^{**}$, 3. $V(p,u) = 1$ and $u \leq w$ imply $V(p,w) = 1$.
V is extended to a valuation on all formulas via the following conditions:

$V(\varphi \wedge (\vee) \psi, w) = 1$ iff $V(\varphi, w) = 1$ and (or) $V(\psi, w) = 1$
$V(\sim\varphi, w) = 1$ iff $V(\varphi, w^*) = 0$

For implication, one distinguishes between starred and unstarred worlds as follows.

For $w \in W$, $V(\varphi \rightarrow \psi, w) = 1$ iff for every $w' \in W$ such that $w \leq w'$, $V(\varphi, w') = 1 \Rightarrow V(\psi, w') = 1$.
For $w \in W^*$, $V(\varphi \rightarrow \psi, w) = 1$ iff $V(\varphi, w^*) = 1 \Rightarrow V(\psi, w) = 1$.

[1] A natural understanding of the $*$ operator is a moot point. For discussion and further links we refer the reader to [5].

A proposition φ is said to be *true* in a Routley model $\mathcal{M} = \langle W, \leq, ^*, V \rangle$, if $V(\varphi, v) = 1$, where v is an arbitrarily selected unstarred element of W. A formula is valid if it is true in every Routley model. It is easy to prove by induction that condition 3 above holds for any formula φ, ie $V(\varphi, u) = 1 \;\; \& \;\; u \leq w \quad \Rightarrow \quad V(\varphi, w) = 1$. Moreover $\mathbf{N}^-$ is complete for the Routley semantics in the sense that a formula is valid iff it is a theorem of $\mathbf{N}^-$.

In the full Nelson logic, $\mathbf{N}$, intuitionistic or weak negation $\neg$ is definable, eg by $\neg\varphi := \varphi \rightarrow \sim\varphi$. In the paraconsistent $\mathbf{N}^-$, however, there are various options for introducing an intuitionistic style negation, $\neg$. We shall consider a second negation $\neg$ defined via the constant $\bot$, $\neg\varphi := \varphi \rightarrow \bot$, characterised semantically via the following condition:

$$V(\bot, w) = 0 \;\; \text{for all} \;\; w.$$

The logic obtained in this way we denote $\mathbf{N}^-_\neg$. Note that validity of $\sim$-free formulas in $\mathbf{N}^-_\neg$ can be defined via unstarred worlds only. Moreover, this definition is identical with the validity of formulas in Kripke semantics for intuitionistic logic. Therefore, $\mathbf{N}^-_\neg$ is a conservative extension of intuitionistic logic.

Proposition 1. $\mathbf{N}^-_\neg = \mathbf{N}^- + \{\bot \rightarrow p, p \rightarrow \sim \bot\}$

Proof sketch. The validity of formulas $\bot \rightarrow p$ and $p \rightarrow \sim \bot$ is verified directly. The completeness follows by an obvious modification of the original Routley proof. □

Thus, the logic $\mathbf{N}^-_\neg$ is identical with the logic $\mathbf{N4}^\bot$ studied in [17] and we can apply to extensions of $\mathbf{N}^-_\neg$ the results of [17].

4 Answer Set Semantics

We consider the version of answer set semantics defined for disjunctive logic programs with two kinds of negation [7]. We use the syntax of $\mathbf{N}^-_\neg$ to describe the rules or formulas of programs. Strong negation is denoted therefore by '$\sim$' and the second, default negation, usually written as 'not' will be denoted by '$\neg$'. The formulas of disjunctive programs therefore have the form

$$L_1 \wedge \ldots \wedge L_m \wedge \neg\, L_{m+1} \wedge \ldots \wedge \neg\, L_n \rightarrow K_1 \vee \ldots \vee K_k \tag{1}$$

where each L_i, K_j is a literal (atom or strongly negated atom) and we may have $m = n$ and m or n may be zero. A logic program Π is a set of such formulas. The set of all literals in the language of Π is denoted by Lit. We assume the reader is familiar with the usual definition of answer set [7] employing the notion of reduct. Paraconsisent answer sets (PAS) are defined, following [30], in the same way but omitting the clause ((ii) of [7]) which essentially imposes consistency by making Lit the only inconsistent answer set.

Some simple examples from [30] may help to illustrate the difference between answer sets and PAS. For instance the program $\{\sim A; \neg B \rightarrow A\}$ has no answer sets but an inconsistent PAS $\{A, \sim A\}$, while the program $\{A; \sim A; \neg B \rightarrow B\}$ has Lit as its only answer sets, but no paraconsistent answer sets. The reader is referred to [30] for further properties of PAS and additional motivation.

5 Here-and-There Models

In the semantics for intermediate or superintuitionistic logics, the so-called logic of *here-and-there* can be captured by rooted frames with two elements, commonly denoted by h and t and called 'here' and 'there', with $h \leq t$. If we work instead in the lattice of extensions of Nelson's logic $\mathbf{N}^-$, we can consider here-and-there models as special kinds of Routley models. In the obvious way therefore a *Routley here-and-there model* can be represented as a Routley model $\mathcal{M} = \langle W \cup W^*, \leq, ^*, V \rangle$, where $W = \{h, t\}$ comprises two worlds 'here' and 'there', such that $h \leq t$, and $W^* = \{h^*, t^*\}$ comprises the "starred" worlds. It follows that $t^* \leq h^*$. We now consider the relation between Routley here-and-there models and answer sets, in particular paraconsistent answer sets. We shall therefore assume that a second negation $\neg$ is present in the language understood according to the above semantics for $\mathbf{N}^-_\neg$. The logic determined by here-and-there Routley models according to the semantics of $\mathbf{N}^-_\neg$ will be denoted by $\mathbf{N}_9$. We prove (Proposition 6) that this logic is determined by a 9-valued matrix, which explains the choice of notation.

Recently a frame based semantics has been presented by Alcantara *et al* [1] to provide a more declarative style of representation for paraconsistent answer sets. Their approach (henceforth ADP frames) is based on here-and-there style worlds and 2-valued truth-valuations. In fact they call their models HT^2-models. Syntax is as for the logic $\mathbf{N}^-_\neg$, but the frames employ no fewer than 3 accessibility relations to model the logical constants $\rightarrow, \neg, \sim$. In addition the authors use point sets and so-called belief sets. It turns out however that several items of this rather complicated machinery are redundant. Once redundancies are removed it is easy to see that the HT^2-models can be replaced without loss by Routley here-and-there models. We now indicate the steps to show this.

5.1 ADP Frames

[1] define frames based on point sets, where a point set $\mathcal{P} = \langle Q, \leq \rangle$, with Q a set and $\leq$ a partial ordering on Q. *Propositions* on $\mathcal{P}$ are upwards closed subsets of Q. Frames are point sets together with acccessibility relations. According to [1], an HT^2 frame is defined as follows. The underlying point set $\mathcal{P} = \langle Q, \leq \rangle$ is such that Q has four elements that we will denote here by h, h^*, t, t^*, where $h \leq t$ and $t^* \leq h^*$. Secondly, three accessibility relations are defined on $\mathcal{P}$ that we will denote here by $R, R_\sim, R_\neg$. R is a ternary relation and $R_\sim$ and $R_\neg$ are binary relations determined by:

$$R(h,h,h), R(h,h,t), R(h,t,t), R(t,t,t), R(t,h,t),$$
$$R(t^*,t,t^*), R(t^*,t,h^*), R(t^*,h,t^*), R(t^*,h,h^*), R(h^*,h,h^*).$$
$$R_\sim(h,h^*), R_\sim(t,t^*), R_\sim(h^*,h), R_\sim(t^*,t), R_\sim(h,t^*), R_\sim(t^*,h).$$
$$R_\neg(h,h), R_\neg(t,t), R_\neg(h,t), R_\neg(t,h), R_\neg(h^*,h), R_\neg(t^*,h), R_\neg(t^*,t).$$

An HT^2 model M is formed from an HT^2 frame by assigning atoms to the four points in accordance with the interpretation that propositions form upwards closed sets. In other words if we denote the set of atoms true in w by W, then we have $w \leq w'$ implies $W \subseteq W'$. The assignment of atoms to points is extended to all propositions via standard conditions for conjunction and disjunction and the following clauses:

$(M, w) \models \varphi \rightarrow \psi$ iff $\forall w', w'' s.t. R(w, w', w''), (M, w') \models \varphi \Rightarrow (M, w'') \models \psi$.
$(M, w) \models \neg\varphi$ iff $\forall w' s.t. R_{\neg}(w, w'), (M, w') \not\models \varphi$
$(M, w) \models \sim\varphi$ iff $\forall w' s.t. R_{\sim}(w, w'), (M, w') \not\models \varphi$

It is straightforward to check that HT^2 models are equivalent to Routley here-and-there models. Let M be an HT^2 model. Then the corresponding Routley model $\mathcal{M} = \langle W, W^*, \leq, ^*, V \rangle$ consists of the same four points or worlds h, h^*, t, t^*, such that for any propositional atom p, and world w, $V(p, w) = 1 \Leftrightarrow (M, w) \models p$, and conversely.

Proposition 2. *For all formulas φ and worlds w, $V(\varphi, w) = 1 \Leftrightarrow (M, w) \models \varphi$.*

Proof sketch. By assumption the two models make the same truth assignment for atoms. The truth conditions for conjunction and disjunction are also the same in each case. For the remaining connectives one simply applies the definitions. The verification is tedious but simple. We illustrate the case of strong negation; other cases are left to the reader. By Routley semantics we have $V(\sim\varphi, w) = 1$ iff $V(\varphi, w^*) = 0$. Consider the case $w = h$. Then we have $V(\sim\varphi, h) = 1$ iff $V(\varphi, h^*) = 0$. By the ADP semantics, $(M, h) \models \sim\varphi$ iff for all w s.t. $R_{\sim}(h, w), (M, w) \not\models \varphi$ iff $(M, h*) \not\models \varphi$ (applying the definition of $R_{\sim}$ and noting that by the hereditary condition and the fact that $t^* \leq h^*$, $(M, h^*) \not\models \varphi$ implies $(M, t^*) \not\models \varphi$). So $V(\sim\varphi, h) = 1 \Leftrightarrow (M, h) \models \sim\varphi$. The argument for other worlds w is analogous. □

We can therefore replace the rather complicated HT^2 models of [1] by simpler Routley here-and-there models. Aside from simplicity, an advantage is that we know that Routley models capture logics in the lattice $\mathcal{E}N^-$ of extensions of Nelson's logic and with minor modification they capture the logic of Fitch and its extensions.

5.2 Minimal Models

Consider Routley here-and-there models $\mathcal{M} = \langle W, W^*, \leq, ^*, V \rangle$, where $W = \{h, t\}$ with $h \leq t$, and $W^* = \{h^*, t^*\}$. More succinctly, represent $\mathcal{M}$ as a pair of ordered pairs $(\langle H, H^* \rangle, \langle T, T^* \rangle)$, where the upper-case letters denote the sets of atoms verified at the corresponding point or world. Even more succinctly we can denote an unstarred, starred pair in the form $\mathbf{H}$ and $\mathbf{T}$, so $\mathcal{M}$ is represented simply as $(\mathbf{H}, \mathbf{T})$, with $\mathbf{H} = \langle H, H^* \rangle$ and $\mathbf{T} = \langle T, T^* \rangle$. Such pairs can be partially ordered as follows. We say in general that $\mathbf{H} \leq \mathbf{T}$ if $H \subseteq T$ and $T^* \subseteq H^*$. Notice that by the Routley semantics, if $(\mathbf{H}, \mathbf{T})$ is a model then necessarily $\mathbf{H} \leq \mathbf{T}$. This ordering can be extended to a partial ordering $\trianglelefteq$ among models as follows. We set $(\mathbf{H}, \mathbf{T}) \trianglelefteq (\mathbf{H}', \mathbf{T}')$ if (i) $\mathbf{T} = \mathbf{T}'$; $(ii)\mathbf{H} \leq \mathbf{H}'$. A model $(\mathbf{H}, \mathbf{T})$ in which $\mathbf{H} = \mathbf{T}$ is said to be *total*.

We are interested here in a special kind of minimal model that we call an equilibrium model. Let Π be a theory. A model $\mathcal{M}$ of Π is said to be an *equilibrium* model of Π if (i) $\mathcal{M}$ is total; (ii) $\mathcal{M}$ is minimal among models of Π under the ordering $\trianglelefteq$. In other words an equilibrium model of Π has the form $(\mathbf{T}, \mathbf{T})$ and is such that if $(\mathbf{H}, \mathbf{T})$ is a model of Π with $\mathbf{H} \leq \mathbf{T}$, then $\mathbf{H} = \mathbf{T}$. The main property of equilibrium models is expressed in the next proposition.

We introduce some notation. For a set X of atoms we set $\sim X = \{\sim A : A \in X\}$ and denote by $\overline{X}$ the complement of X. Then, note that a literal L is true in a model $(\mathbf{H}, \mathbf{T})$, just in case $L \in H \cup \sim\overline{H^*}$.

Proposition 3. *Let Π be a disjunctive logic program. A set of literals S is a paraconsistent answer set of Π just in case it can be represented in the form $S = T \cup \sim\overline{T^*}$ where T and T^* are sets of atoms, and $(\mathbf{T}, \mathbf{T})$ is an equilibrium model of Π such that $\mathbf{T} = \langle T, T^* \rangle$.*

Proof sketch. By Proposition 2, HT^2 models can be equivalently represented as Routley here-and-there models $(\mathbf{H}, \mathbf{T})$. Using this representation, by Theorem 1 of [1], $(\mathbf{H}, \mathbf{T})$ is a model of a disjunctive program Π iff the sets $I = H \cup \sim\overline{H^*}$ and $J = T \cup \sim\overline{T^*}$ are models of the reduct Π^J. By Theorem 2 of [1], paraconsisent answer sets correspond to total models $(\mathbf{T}, \mathbf{T})$ in which $\mathbf{T}$ satisfies a certain minimality condition. By this condition, if $(\mathbf{H}, \mathbf{T})$ is a model of Π different from $(\mathbf{T}, \mathbf{T})$ but with $\mathbf{H} \leq \mathbf{T}$, then clearly $I = H \cup \sim\overline{H^*}$ is a model of Π^J strictly contained in $J = T \cup \sim\overline{T^*}$. But this is impossible by the definition of answer sets. Hence paraconsistent answer sets correspond to total models that are minimal under $\trianglelefteq$. □

This shows the equivalence of paraconsistent answer sets and equilibrium models in $\mathbf{N}_9$. However, notice that the notion of equilibrium model is defined for arbitrary propositional theories, not only disjunctive logic programs. From the above proposition it follows that any theories logically equivalent in $\mathbf{N}_9$ have the same equilibrium models. In general we say that two programs or theories, Π_1 and Π_2, are *strongly equivalent* if for any set of formulas Π, $\Pi_1 \cup \Pi$ and $\Pi_2 \cup \Pi$ have the same equilibrium models. So clearly any two theories equivalent in $\mathbf{N}_9$ are strongly equivalent. As in the ordinary (non-paraconsistent) case, the converse also holds.

Proposition 4. *Two propositonal theories, Π_1 and Π_2, are strongly equivalent if and only if they are equivalent in $\mathbf{N}_9$.*

The method of proof is similar to that of Theorem 1 of [13] adapted to take account of strong negation and paraconstency. We omit the details.

Finally, to obtain ordinary answer sets for the non-paraconsistent case we add one more condition on Routley models, namely we require for all worlds w, $w \leq w^*$. The notion of equilibrium is defined exactly as before and the resulting here-and-there Routley models are axiomatised by the logic $\mathbf{N}_5$, studied in [11,22,23].

6 On the Logic $\mathbf{N}_9$

In this section we axiomatise the logic $\mathbf{N}_9$, prove that it can be defined via a 9-element matrix and establish that it possesses the Craig interpolation property. To this end we will use some facts from [17] concerning the algebraic semantics of $\mathbf{N}^-_\neg$ and the interrelations of the lattices of extensions $\mathcal{E}\mathbf{N}^-_\neg$ and $\mathcal{E}Int$, where Int is intuitionistic logic. The lattice of logics $\mathcal{E}\mathbf{N}^-_\neg$ ($\mathcal{E}Int$) consists of sets of formulas in the respective language containing all $\mathbf{N}^-_\neg$-(Int-)tautologies and closed under the rules of substitution and *modus ponens*, ie. a logic is identified with its set of tautologies. The meet-operation in these lattices coincides with set-theoretical intersection and the join operation is the closure of the sum of logics wrt substitution and *modus ponens*. If $L \in \mathcal{E}\mathbf{N}^-_\neg(\mathcal{E}Int)$ and X is a set of formulas, then $L + X$ denotes the least logic containing L and X.

Recall that the algebraic semantics for Int and its extensions is provided by Heyting algebras. An algebraic structure $\mathcal{A} = \langle A, \vee, \wedge, \to, 0, 1\rangle$ is said to be a Heyting algebra if its reduct $\langle A, \vee, \wedge, 0, 1\rangle$ is a bounded lattice and the pseudo-complement $a \to b$ is the greatest x such that $a \wedge x \leq b$. Validity of formulas in Heyting algebras ($\mathcal{A} \models \varphi$) is understood in a usual way assuming that the unit 1 is the only distinguished element.

Intuitionistic logic Int coincides with the set of formulas valid in all Heyting algebras. For any Heyting algebra $\mathcal{A}$ define the logic $L\mathcal{A} := \{\varphi : \mathcal{A} \models \varphi\}$. Clearly, $L\mathcal{A} \in \mathcal{E}Int$. For a class $\mathcal{K}$ of Heyting algebras, $L\mathcal{K} = \bigcap\{L\mathcal{A} : \mathcal{A} \in \mathcal{K}\}$. If $L = L\mathcal{K}$ we say that L is characterized by the class $\mathcal{K}$. Every element of $\mathcal{E}Int$ is characterized by some class of Heyting algebras.

6.1 Twist-Structures

Definition 1. *Let $\mathcal{A} = \langle A, \vee, \wedge, \to, 0, 1\rangle$ be a Heyting algebra. A* full twist-structure *over $\mathcal{A}$ is an algebra $\mathcal{A}^{\bowtie} = \langle A \times A, \vee, \wedge, \to, \sim, \bot, 1\rangle$ with twist-operations defined for $(a, b), (c, d) \in A \times A$ as follows:*

$$(a, b) \vee (c, d) := (a \vee c, b \wedge d),\ (a, b) \wedge (c, d) := (a \wedge c, b \vee d)$$
$$(a, b) \to (c, d) := (a \to c, a \wedge d),\ \sim (a, b) := (b, a),$$
$$\bot := (0, 1), 1 := (1, 0).$$

A twist-structure *over $\mathcal{A}$ is an arbitrary subalgebra $\mathcal{B}$ of the full twist-structure $\mathcal{A}^{\bowtie}$ such that $\pi_1(\mathcal{B}) = \mathcal{A}$ (in which case also $\pi_2(\mathcal{B}) = \mathcal{A}$), where π_i, $i = 1, 2$, is a projection of a direct product onto the ith coordinate.*

For a Heyting algebra $\mathcal{A}$ denote by $\mathcal{A}_0^{\bowtie}$ a twist-structure over $\mathcal{A}$ with the universe

$$|\mathcal{A}_0^{\bowtie}| = \{(a, b) : a, b \in \mathcal{A}, a \wedge b = 0\}.$$

A valuation into a twist-structure $\mathcal{B}$ is defined in a usual way as a homomorphism of an algebra of formulae into $\mathcal{B}$. The relation $\mathcal{B} \models_{\bowtie} \varphi$, where φ is a formula of a respective language, means that $\pi_1 v(\varphi) = 1$ for any $\mathcal{B}$-valuation v. For a formula φ, the relation $\models_{\bowtie} \varphi$ means that $\mathcal{B} \models_{\bowtie} \varphi$ for any twist-structure $\mathcal{B}$.

Theorem 1. [17] *Let φ be a formula. Then $\mathbf{N}_{\neg}^{-} \vdash \varphi \Leftrightarrow \models_{\bowtie} \varphi$.*

Moreover, it was proved in [17] that algebras isomorphic to twist-structures form a variety and that any logic extending $\mathbf{N}_{\neg}^{-}$ is characterized by some subvariety of this variety.

For a logic $L \in \mathcal{E}Int$, define the $\mathbf{N}_{\neg}^{-}$-extensions $\eta^{-}(L)$ and $\eta(L)$ as follows

$$\eta^{-}(L) = \mathbf{N}_{\neg}^{-} + L \text{ and } \eta(L) = \eta^{-}(L) + \sim p \to (p \to q).$$

Note that the logic $\eta(Int) = \mathbf{N}_{\neg}^{-} + \sim p \to (p \to q)$ can be identified with the logic $\mathbf{N}$, because if we define in $\mathbf{N}$ the new constant $\bot := \sim (p \to p)$ it satisfies the two new axioms of the logic $\mathbf{N}_{\neg}^{-}$. It was proved in [8] that $\eta(L)$ is the least conservative extension of L in the lattice $\mathcal{E}\mathbf{N}$. In [17], it was stated that $\eta^{-}(L)$ is the least conservative extension of L in the lattice $\mathcal{E}\mathbf{N}_{\neg}^{-}$.

For any twist-structure $\mathcal{A}$ define the logic $L\mathcal{A} := \{\varphi : \mathcal{A} \models_{\bowtie} \varphi\}$. Then $L\mathcal{A} \in \mathcal{E}\mathbf{N}_{\neg}^{-}$.

Proposition 5. *Let $\mathcal{A}$ be a Heyting algebra and $L = L\mathcal{A}$.*

1. [17] $\eta^{-}(L) = L\mathcal{A}^{\bowtie}$.
2. [8] $\eta(L) = L\mathcal{A}_0^{\bowtie}$.

6.2 Axioms of $\mathbf{N_9}$

It it well known that the logic HT of here-and-there is determined by the two-element Kripke frame or by the three-element Heyting algebra $\mathbf{3}$, where $|\mathbf{3}| = \{0, 1/2, 1\}$, $0 \leq 1/2 \leq 1$. We prove now that $\mathbf{N}_9$ is determined by the twist-structure $\mathbf{3}^{\bowtie}$.

Proposition 6. $\mathbf{N}_9 = L\mathbf{3}^{\bowtie}$.

Proof. Let $\mathcal{M} = (\langle H, H^* \rangle, \langle T, T^* \rangle)$ be a Routley here-and-there model. Define a $\mathbf{3}^{\bowtie}$ valuation $v_{\mathcal{M}}$ by the following equivalences.

$$\begin{array}{ll} \pi_1 v_{\mathcal{M}}(p) = 0 \text{ iff } p \notin T & \pi_1 v_{\mathcal{M}}(p) = 1/2 \text{ iff } p \in T \setminus H \\ \pi_1 v_{\mathcal{M}}(p) = 1 \text{ iff } p \in H & \pi_2 v_{\mathcal{M}}(p) = 0 \text{ iff } p \notin H^* \\ \pi_2 v_{\mathcal{M}}(p) = 1/2 \text{ iff } p \in H^* \setminus T^* & \pi_2 v_{\mathcal{M}}(p) = 1 \text{ iff } p \in T^* \end{array}$$

Clearly, the mapping $\mathcal{M} \mapsto v_{\mathcal{M}}$ establishes a one-to-one correspondence between Routley here-and-there models and $\mathbf{3}^{\bowtie}$-valuations. An easy induction on the structure of formulas shows that $\pi_1 v_{\mathcal{M}}(\varphi) = 1$ iff φ is true at h in $\mathcal{M}$ and so at every unstarred world of $\mathcal{M}$. □

Taking into account Proposition 5 we arrive at

Corollary 1. $\mathbf{N}_9 = \eta^{-}(HT)$.

The logic of here-and-there is axiomatized over Int by the formula

$$p \vee (p \to q) \vee \neg q.$$

Therefore, $\mathbf{N}_9$ is axiomatized by the same formula over $\mathbf{N}_{\neg}^{-}$. Note also that $\mathbf{N}_5 = \eta(HT)$ and it is characterized by the five-element twist-structure $\mathbf{3}_0^{\bowtie}$, which explains the choice of notation.[2]

6.3 Interpolation Property

We say that a logic L possesses the *Craig interpolation property*, for short CIP, if $\varphi \to \psi \in L$ implies that there exists a formula χ such that $\varphi \to \chi \in L$ and $\chi \to \varphi \in L$, and χ has occurences of common variables of φ and ψ only.

The algebraic counterpart of this property is as follows. Let $\mathcal{K}$ be a class of algebras. We say that $\mathcal{K}$ has an *amalgamation property* if for any algebras $\mathcal{A}_0, \mathcal{A}_1, \mathcal{A}_2 \in \mathcal{K}$ and monomorphisms $i_1 : \mathcal{A}_0 \hookrightarrow \mathcal{A}_1$ and $i_2 : \mathcal{A}_0 \hookrightarrow \mathcal{A}_2$, there exists an algebra $\mathcal{A} \in \mathcal{K}$ and monomorphisms $\varepsilon_1 : \mathcal{A}_1 \hookrightarrow \mathcal{A}$ and $\varepsilon_2 : \mathcal{A}_2 \hookrightarrow \mathcal{A}$ such that $\varepsilon_1 i_1 = \varepsilon_2 i_2$. The triple $(\mathcal{A}, \varepsilon_1, \varepsilon_2)$ is called an *amalgam* of $\mathcal{A}_1$ and $\mathcal{A}_2$ over $\mathcal{A}_0$.

It was proved in [15] that an intermediate logic possesses CIP if and only if the variety of its models has an amalgamation property. The next statement can be obtained by an easy modification of the proof from [15].

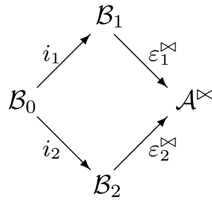
[2] In [1] the authors also mention that propositions on their frames take 9 possible truth values.

Proposition 7. *Let* $L \in \mathcal{E}\mathbf{N}_\neg^-$. *L possesses* CIP *if and only if the family of twist-structures* $\mathcal{A}$ *such that* $\mathcal{A} \models_{\bowtie} L$ *has an amalgamation property.*

Proposition 8. *The logic* $\mathbf{N}_9$ *possesses* CIP.

Proof. Recall that $\mathbf{N}_9 = \eta^-(HT)$, and that models of $\mathbf{N}_\neg^-$ are exactly twist-structures over HT-models. In [15] it was proved that the latter possess the amalgamation property. We verify that the same holds for $\mathbf{N}_9$-models.

Let $\mathcal{A}_0, \mathcal{A}_1, \mathcal{A}_2 \models HT$. Consider twist-structures $\mathcal{B}_0$, $\mathcal{B}_1$ and $\mathcal{B}_2$ over $\mathcal{A}_0$, $\mathcal{A}_1$ and $\mathcal{A}_2$, respectively, and their embeddings $i_1 : \mathcal{B}_0 \hookrightarrow \mathcal{B}_1$ and $i_2 : \mathcal{B}_0 \hookrightarrow \mathcal{B}_2$. Then $\pi_1 i_1$ embeds $\mathcal{A}_0$ into $\mathcal{A}_1$, and $\pi_1 i_2$ embeds $\mathcal{A}_0$ into $\mathcal{A}_2$. There is an amalgam $(\mathcal{A}, \varepsilon_1, \varepsilon_2)$ of $\mathcal{A}_1$ and $\mathcal{A}_2$ over $\mathcal{A}_0$. Lift up the monomorphisms ε_1 and ε_2 as follows. Put $(\varepsilon_i)^{\bowtie}((a,b)) = (\varepsilon_i(a), \varepsilon_i(b))$ for $(a,b) \in \mathcal{B}_i$. It is not hard to check that $(\varepsilon_1)^{\bowtie}$ and $(\varepsilon_2)^{\bowtie}$ are embeddings of $\mathcal{B}_1$ and $\mathcal{B}_2$ into $\mathcal{A}^{\bowtie}$. So we obtain the following diagram.



Commutativity of this diagram immediately follows from the equalities $\varepsilon_1 \pi_1 i_1 = \varepsilon_2 \pi_2 i_2$. □

Note that CIP for $\mathbf{N}_5$ can be established in the same way.

7 Conclusions and Future Work

Extending the results of [19] and [1] we have shown how both ordinary and paraconsistent answer sets can be captured via possible worlds models due to Routley [29]. In the case of PAS, the underlying logic, $\mathbf{N}_9$, has been identified axiomatically and algebraically, and an important metalogical property - interpolation - has been proved. A consequence of our analysis is that PAS can easily be defined for arbitrary theories. An interesting feature to emerge is that the underlying logic of PAS, although paraconsistent, still extends intuitionistic inference (unlike say well-founded semantics), and is still a conservative extension of the logic of here-and-there.

Answer set programming is already being applied in areas such as information integration where data sources may be incomplete and inconsistent (see INFOMIX: http://sv.mat.unical.it/infomix/). In this area paraconsistent reasoning may be a useful extension to ordinary ASP, and here PAS may be worthy of further study.[3] Following Wagner [34] one can distinguish different styles of information processing in the presence of contradictions. Our characterisation of PAS should provide a useful basis

[3] Currently INFOMIX uses various database repair techniques to restore consistency prior to query evaluation.

for developing these issues further in the setting of logic programming. For example we may want to assume explosiveness for one kind of statement and not for another; eg one might distinguish essential information, where contradictions are not admitted, from ”information noise”, where contradictions are irrelevant. We can also discuss different ways in which contradictions may be localised. In this respect, the logic $\mathbf{N}^{-}$ has the advantage of having no contradictory extensions [17]. It is not compatible with contradiction as a scheme, ie. it admits only local contradictions. Another important avenue for further research is the comparison of PAS with other approaches to paraconsistency in logic programming and nonmonotonic reasoning. In particular our purely logical characterisation of PAS may facilitate comparison with paraconsistent logics in general.

References

1. J. Alcantara, C. Damásio & L. M. Pereira. A Declarative Characterisation of Disjunctive Paraconsistent Answer Sets. In R. López de Mántaras & L. Saitta (eds), *Proc. of ECAI 2004*, IOS Press, 2004, 951-952. Full version available at http://centria.di.fct.unl.pt/ jfla/publications/.
2. A. Almukdad & D. Nelson. Constructible Falsity and Inexact Predicates. *J. Symbolic Logic* 49 (1984), 231-233.
3. H. Blair & V.S. Subrahmanian. Paraconsistent logic programming. *Theoretical Computer Science* 68 (1989), 135-154.
4. C. Damásio & L. M. Pereira. A survey of paraconsistent semantics for logic programs. in D. Gabbay & P. Smets (eds), *Handbook of Defeasible Reasoning and Uncertaintly Management Systems, Vol. 2*, Kluwer, 1998, 241-320.
5. J.M. Dunn. Relevance logic and entailment. in D. Gabbay et al. (eds), *Handbook of philosophical logic. Vol. III: Alternatives to classical logic,* Reidel, Synth. Libr. 166, 1986, 117-224.
6. F. B. Fitch. *Symbolic Logic, an Introduction*. Ronald Press, New York, 1952.
7. M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9 (1991), 365–385, .
8. V. Goranko. The Craig Interpolation Theorem for Propositional Logics with Strong Negation. *Studia Logica* 44 (1985), 291-317.
9. Y. Gurevich. Intuitionistic logic with strong negation. *Studia Logica*, 36 (1977), 49–59.
10. J. Jaspers. *Calculi for Constructive Communication*. ILLC Dissertation Series 1994-4, ILLC 1994.
11. M. Kracht. On extensions of intermediate logics by strong negation. *Journal of Philosophical Logic*, 27 (1998), 49–73.
12. F. von Kutschera. *Der Satz vom ausgeschlossenen Dritten.* de Gruyter, Berlin, 1985.
13. V. Lifschitz, D. Pearce & A. Valverde. Strongly equivalent logic programs. *ACM Transactions on Computational Logic*, 2 (2001), 526–541.
14. E.G.K López-Escobar. Refutability and Elementary Number Theory. *Indag. Math.* 34 (1972), 362-374.
15. L.L. Maksimova. Craig theorem in super-intuitionistic logics and amalgamated varieties of pseudo-boolean algebras. *Algebra i Logika* 16 (1977), 643–681 (in Russian).
16. D. Nelson. Constructible falsity. *Journal of Symbolic Logic*, 14 (1949),16–26.
17. S.P. Odintsov. The class of extensions of Nelson's paraconsistent logic. *Studia Logica*, 80 (2005), 291–320.

18. D. Pearce. Answer Sets and Constructive Logic. Part II: extended logic programs and related nonmonotonic formalisms. In L. M. Periera & A. Nerode (eds), *Logic Programming and Non-monotonic Reasoning*, MIT press, 1993, pp. 457-475.
19. D. Pearce. A new logical characterization of stable models and answer sets. In *Proc. of NMELP 96*, LNCS 1216, Springer, 1997, pp. 57–70.
20. D. Pearce. From here to there: Stable negation in logic programming. In Dov Gabbay and Heinrich Wansing, eds., *What is Negation?*, Kluwer Academic Pub., 1999, pp. 161–181.
21. D. Pearce. Simplifying logic programs under answer set semantics. In V. Lifschtiz & B. Demoen (eds), *Proc. of ICLP04*. Springer LNCS 3132, 2004, pp. 210–224.
22. D. Pearce, I.P. de Guzmán, and A. Valverde. Computing equilibrium models using signed formulas. In *Proc. of CL2000*, Springer LNCS 1861, 2000, pp. 688–703.
23. D. Pearce, I.P. de Guzmán, and A. Valverde. A tableau calculus for equilibrium entailment. In *Proc. of TABLEAUX 2000*, Springer LNAI 1847, 2000, pp. 352–367.
24. D. Pearce and G. Wagner. Reasoning with Negative Information I: Strong Negation in Logic Programs. In L. Haaparanta, M. Kusch and I. Niiniluoto (eds), *Language, Knowledge and Intentionality*, Acta Philosophica Fennica 49, Helsinki, 1990
25. D. Pearce and G. Wagner. Logic Programming with Strong Negation. In P. Schroeder-Heister (ed), *Extensions of Logic Programming*, Springer LNAI 475, 1991.
26. D. Pearce and A. Valverde. Uniform equivalence for equilibrium logic and logic programs. In V. Lifschitz & I. Niemelä (eds), *Proc. of LPNMR'04*, Springer LNAI 2923, 2004, pp. 194–206.
27. D. Pearce and A. Valverde. Synonymous theories in answer set programming and equilibrium logic. In R. López de Mántaras & L. Saitta (eds), *Proc. of ECAI 2004*, IOS Press, 2004, 388-392.
28. H. Rasiowa. *An Algebraic Approach to Non-classical Logic* PWN, Warwaw and North-Holland, Amsterdam, 1974.
29. R. Routley. Semantical Analyses of Propositional Systems of Fitch and Nelson. *Studia Logica* 33 (1974), 283-298.
30. C. Sakama & K. Inoue Paraconsistent Stable Semantics for Extended Disjunctive Programs *J. Logic & Computation* 5 (1995), 265-285.
31. R. H. Thomason. A Semantical Study of Constructible Falsity. *Zeitschrift für Mathematische Logik* 15 (1969), 247-257.
32. N. N. Vorob'ev. A constructive propositional calculus with strong negation (in russian). *Doklady Akademii Nauk SSR*, 85 (1952), 465–468.
33. N. N. Vorob'ev. The problem of deducibility in constructive propositional calculus with strong negation (in russian). *Doklady Akademii Nauk SSR*, 85 (1952), 689–692.
34. G.Wagner. Ex contradictione nihil sequitur. *Artificial intelligence, IJCAI-91, Proc. 12th Int. Conf. Sydney/Australia 1991*, 1991, pp. 538–543.
35. G. Wagner. *Vivid Logic. Knowledge-Based reasoning with Two Kinds of Negation* Springer LNAI 764, 1994.
36. H. Wansing. *The Logic of Information Structures* Springer LNAI 681, 1993.

The Well Supported Semantics for Multidimensional Dynamic Logic Programs*

F. Banti[1], J.J. Alferes[1] A. Brogi[2], and P. Hitzler[3]

[1] CENTRIA, Universidade Nova de Lisboa, Portugal
jja|banti@di.fct.unl.pt
[2] Dipartimento di Informatica, Università di Pisa, Italy
brogi@di.unipi.it
[3] AIFB, Universität Karlsruhe, Germany
hitzler@aifb.uni-karlsruhe.de

Abstract. *Multidimensional dynamic logic programs* are a paradigm which allows to express (partially) hierarchically ordered evolving knowledge bases through (partially) ordered multi sets of logic programs and allowing to solve contradictions among rules in different programs by allowing rules in more important programs to reject rules in less important ones. This class of programs extends the class of *dynamic logic program* that provides meaning and semantics to *sequences of logic programs*. Recently a semantics named *refined stable model semantics* has fixed some counterintuitive behaviour of previously existing semantics for dynamic logic programs. However, it is not possible to directly extend the definitions and concepts of the refined semantics to the multidimensional case and hence more sophisticated principles and techniques are in order. In this paper we face the problem of defining a proper semantics for multidimensional dynamic logic programs by extending the idea of well supported model to this class of programs and by showing that this concept alone is enough for univocally characterizing a proper semantics. We then show how the newly defined semantics coincides with the refined one when applied to sequences of programs.

1 Introduction

In recent years some effort was devoted to explore the problem of how to update knowledge bases represented by logic programs (LPs) with new rules. This allows, for instance, to better use LPs for representing and reasoning with knowledge that evolves in time, as required in several fields of application. The LPs updates framework has been used, for instance, as the base of the MINERVA agent architecture [14] and the action description language EAPs [3].

Different semantics have been proposed [1,2,5,6,12,15,17,19,18] that assign meaning to arbitrary finite sequences $P_1, \ldots, P_m$ of logic programs, usually called *dynamic logic programs* (DyLPs), each program in the sequence representing a

* This work was supported by project FLUX POSI/40958/SRI/01, and by the European Commission within the 6th Framework Pr. project Rewerse, no. 506779.

C. Baral et al. (Eds.): LPNMR 2005, LNAI 3662, pp. 356–368, 2005.
© Springer-Verlag Berlin Heidelberg 2005

supervenient state of the world. The different states may represent different time points, in which case P_1 is an initial knowledge base, and the other P_is are subsequent updates of the knowledge base or as knowledge coming from different sources that are (totally) ordered according to some precedence, or as different hierarchical instances where the subsequent programs represent more specific information. The role of the semantics of DyLPs is to employ the relationships among different states to precisely determine the meaning of the combined program comprised of all individual programs at each state. Intuitively, one can add, at the end of the sequence, newer rules or rules with precedence (arising from newly acquired, more specific or preferred knowledge) leaving to the semantics the task of ensuring that these added rules are in force, and that previous or less specific rules are still valid (by inertia) only as far as possible, i.e. that they are kept as long as they are not rejected. A rule is rejected whenever it is in conflict with a newly added one (*causal rejection of rules*). Most of the semantics defined for DyLPs [1,2,5,6,12,15,17,19,18] are based on such a concept of causal rejection. *Multidimensional dynamic logic programs* (MDyLPs) [13] generalize DyLPs by considering, instead of sequences, *partially ordered* multisets of programs. This generalization allows to combine in a single framework the possibility of having hierarchically ordered knowledge bases, with evolution in time. While most of the existing semantics[1] for DyLPs coincide on a large class of program updates (cf. [6,11]), there are situations in which the set of (dynamic) stable models (SMs) differs from one semantics to the other. Usually such counter-examples show a counterintuitive behaviour of the semantics when dealing with particular kinds of recursive dependencies. Also the existing semantics for MDyLPs exhibit such counterintuitive behaviour as it emerges from the following example which also illustrates a possible usage of MDyLPs.

Example 1. Roughly speaking, a *joint venture* is a society of companies that is administrated by a delegated administrator. The administrator can pursue his own policy independent of the requests of the partners, though taking the directives of the partners into account. Suppose that two companies α and β constitute a joint venture j whose customers are other companies. The two companies give some directives to the administrator, represented by programs $P_{\alpha 1}$ and $P_{\beta 1}$. The administrator has his own policy which we represent by program P_{j1}. Moreover, both the companies and the administrator update their policies from time to time. We represent such updates, respectively, by programs $P_{\alpha t}$, $P_{\beta t}$ and P_{jt} where t is a time-stamp. Since the policy of the administrator has precedence over the directives of α and β, we assign precedence to the P_{js} over the other programs. At any given time t_k, the decisions of the administrator are described by the semantics of the MDyLP computed at the most recent node P_{jt}. The administrator respects a specific directive from a partner unless it con-

[1] In this paper we restrict our study to semantics generalizing the stable models semantics and that are based on causal rejection of rules. Semantics not based on causal rejection like the ones defined in [17,19], or semantics which use more general forms of rejection like the one presented in [18] are outside the scope of this paper, and are only briefly mentioned in the conclusions.

flicts either with a more recent directive of the same partner or with any of the clauses representing the updates. For any order x, if $cost(x, z)$ is true then z represents the estimated expenses that the j should face in order to satisfy the commission x. Let us assume that the directive of β is to not decline any order whose cost is within a given limit C. This is encoded by the single rule program: $P_{\beta 1}$: $not\, decline(X) \leftarrow cost(X, Z), Z \leq C$. On the other hand, α assigns a degree of reliability to some customers and wants to decline any commission asked by a customer who is not reliable enough. So, $P_{\alpha 1}$ contains:

$$decline(X) \leftarrow comm(X, Y),\ reliability(Y, K),\ rel_limit(Z),\ K \leq Z.$$

plus a database of facts $reliability(C, K)$ for customers C, and a fact $rel_limit(li)$ for representing the limit under which a customer is considered to be unreliable. The initial policy of the administrator is to accept any commission that is acceptable and not declined, having also a set of rules for establishing when a commission is acceptable. Moreover, whenever several orders are received from the same costumer, if one order is accepted, then the others can not be declined. So, P_{j1} contains some rules for $acceptable(X)$ plus:

$$accept(X) \leftarrow acceptable(X), not\, decline(X). \tag{1}$$

$$not\, decline(X) \leftarrow accept(K), comm(X, Y), comm(K, Y). \tag{2}$$

Let us suppose a commission $x1$ arrives from the company y, the commission is acceptable, and the reliability of y is high enough. Hence the order is accepted. At time t_2 the administrator updates his policy by deciding that no commission from the customer y will be declined. This is done by updating P_{j1} with the program P_{j2} with the rule: $not\, decline(X) \leftarrow comm(X, y)$.

At time t_3, α augments the limit of reliability. This is done by the update $P_{\alpha 3}$ with the two facts $not\, rel_limit(li)$ and $rel_limit(new_li)$. Suppose that at time t_4 a commission x_2 arrives from y and the reliability of y is under the new limit of reliability. The directives in P_{α_1} say to decline the commission. Nevertheless the policy of the administrator encoded by the clause in P_{j2} conflicts with this directive. Since P_{j2} has precedence over $P_{\alpha 1}$, the latter is rejected and the order is accepted. Note how the partially ordered programs are used to represent precedence among rules coming from different sources, as well as to represent updates of rules.

So far we provided an example of how multidimensional dynamic logic programs work. We show now a problematic situation involving cyclic dependencies. At time t_5, another commission x_3 arrives from a new costumer $y2$. The reliability of $y2$ is below the current limit, but the estimated cost of x_3 is below the limit C and, moreover x_3 is acceptable. Hence the directives of α and β collide. Since the directives of the two partners are not comparable, none can overcome the other. From an intuitive point of view the policy of j should not allow to judge whether the order should be declined or not. Hence, intuitively the semantics of the considered $MDyLP$ should allow the administrator to detect the conflict and specify, by a new update, whether to decline the new order or accept it.

Unfortunately, none of the existing semantics for MDyLPs matches such intuition. The rules 1 and 2 have cyclic instances for $X = K$. Such a cycle is not relevant in the other examined cases. For $X = K = x_3$, instead, the rule 2 rejects the rule in $P_{\alpha 1}$ thus allowing the semantics to have a model where the commission x_3 is not declined.

It is clearly possible to find also much more complex examples involving complicated self dependencies among rules. Similar examples are known also in the case of DyLPs. In [1] the authors propose the *refined extension principle*,which should be satisfied by a proper semantics for DyLPs in order to avoid such counterintuitive behaviour and then proposed the *refined stable model semantics for DyLPs* that complies with such a principle. Unfortunately, as we show in section 3, the definition of the refined semantics cannot be extended directly to MDyLPs. Moreover, the refined extension principle is too weak for uniquely determining one "right" semantics. For example, the trivial semantics that assigns to each DyLP the empty set of models, satisfies the principle, which is obviously unsatisfactory. We hence need stronger new criteria and techniques.

We begin this paper by, after recalling preliminary notions in Section 2, providing insights of the existing semantics for DyLPs and MDyLPs and explain why a new approach to the problem, based on stronger criteria, is in order. Then, in Section 4 we introduce such a criterium by extending the notion of *well-supported model* (WS model) [4,7] to DyLPs and MDyLPs. Fages [7] shows the equivalence between the concept of stable model and WS model, i.e. given a program P, an interpretation M is a stable model of P iff it is a WS model of P. By extending the definition of well supported model to MDyLPs, we obtain a new semantics for such class of programs. We also show how well supported models do not show counterintuitive behaviour in the illustrated example and, moreover, they provide new insights of the matrix of the behaviour of the other semantics for MDyLPs. Finally, we show that the well supported model semantics coincides with the refined semantics of [1] if we restrict to sequences of programs. For this reason we refer to the defined semantics also as the *refined semantics for MDyLPs*. Well-supported models do already provide a semantics for MDyLPs. Such a descriptive characterization, however, is not completely satisfactory for several reasons, the main one being the problem of finding a reasonable algorithm for its computation. So, in Section 5, we provide an alternative, though equivalent and more traditional characterization based on a fixpoint operator. We then establish relationships between the well supported semantics and the existing semantics for MDyLPs, and show that any WS model is also a model in the existing semantics.

2 Background and Preliminaries

In the following, a *propositional language* $\mathcal{L}$ is a (possibly countably infinite) set of atoms. A *literal in* $\mathcal{L}$ is an atom A of $\mathcal{L}$ or the (default) negation *not* A of an atom of $\mathcal{L}$. We say that A is the *default complement* of *not* A and vice versa. Given a set of literals I, we say a literal L is *true* (resp. *false*) in I iff $L \in I$

(resp. $not\, L \in I$). In the sequel a (two-valued) *interpretation* is a set of literals of $\mathcal{L}$ such that for each atom $A \in \mathcal{L}$ *exactly one* of A or $not\, A$ belongs to I. To simplify the notation, whenever it is clear that we are talking about two-valued interpretations we omit all its negative literals. Let $\mathcal{L}$ and $\mathcal{L}'$ be two languages such that $\mathcal{L} \subset \mathcal{L}'$. Let M be an interpretation over $\mathcal{L}'$. We use the notation $M|_{\mathcal{L}}$ for the set of literals of M in $\mathcal{L}$. Given two interpretations M and M^* over $\mathcal{L}'$, we use the notation $M \equiv |_{\mathcal{L}} M^*$ for $M|_{\mathcal{L}} \equiv M^*|_{\mathcal{L}}$.

2.1 Well-Supported Models

The semantic analysis which we make in this paper rests on the notion of *level mapping* over a set of atoms $\mathcal{L}$, where a *level mapping* ℓ is a function from $\mathcal{L}$ to the set of natural numbers. We also lift ℓ to negative literals of the form $not\, A$, where A is an element of $\mathcal{L}$, by setting $\ell(not\ A) = \ell(A)$. Given a conjunction of literals $C = L_1, \ldots, L_n$ we further extend ℓ by assigning to C the value $\ell(L_i)$, where i is chosen such that the value of $\ell(L_i)$ is maximal, i.e. $\ell(C) = \max(\{\ell(L_i) : L_i \in C\})$. For convenience, and by slight abuse of notation, we assign the value -1 to the empty conjunction of literals. Our approach is stimulated by recent results on uniform characterizations of different semantics for LPs in terms of level mappings as introduced in [10] and extended in [9]. This perspective provides an additional tool and guidelines on how to obtain reasonable new semantics for new classes of programs.

A *normal logic program* over a language $\mathcal{L}$ is any (possibly countably infinite) set of rules of the form $A \leftarrow body$, where A is any atom of $\mathcal{L}$ and *body* is any conjunction of literals of $\mathcal{L}$. Several different (two-valued) semantics are being used for assigning meaning to programs, including the *supported model semantics* [4], the *minimal supported model semantics* [4] and the *stable model semantics* [8]. Given any program P, the set of all supported models ($SU(P)$), the set of all minimal supported models ($MSU(P)$) and the set of all stable models ($SM(P)$) of P are related by $SU(P) \supseteq MSU(P) \supseteq SM(P)$. For large classes of programs these sets of models coincide, but there are particular cases where the inclusions above are strict.

Example 2. Consider the program $P : \ A \leftarrow A$. P has unique SM $\emptyset$ which coincides with the unique minimal supported model. It has $\{A\}$ as a second supported model. All the cited semantics have $\emptyset$ as unique model of the program consisting of the empty set of clauses. Hence, for the supported model semantics adding tautologies of the form $A \leftarrow A$ to a program may change its semantics.

Example 3. Consider now $P_1 : A \leftarrow not\, A. \quad A \leftarrow A$. P_1 has no stable models but has $\{A\}$ as unique minimal supported model. The program $P_2 :\ A \leftarrow not\, A$, has no minimal supported model. Hence, again, the introduction of the tautology $A \leftarrow A$ has changed the semantics of the program.

Stable models for normal LPs can be characterized in terms of level mappings, and in this disguise they are termed *well-supported models* [7]. A model is well-supported iff it is possible to define a level mapping over the literals of the

language, such that a literal A belongs to the model iff there is a rule in the program whose head is A, whose body is true in the considered model and the level of A is greater than the level of any atom in the body. Formally:

Definition 1. *Let P be a normal logic program over the language $\mathcal{L}$. An interpretation M over $\mathcal{L}$ is a* well-supported model of P *iff i) M is a model of P and ii) there exists a level mapping ℓ defined over $\mathcal{L}$, such that for each atom A in M there exists a rule $A \leftarrow A_1, \ldots, A_n, not\, B_1, \ldots, not\, B_m$ with $M \models A_1, \ldots, A_n, not\, B_1, \ldots, not\, B_m$ and $\ell(A) > \ell(Ai)$ for each A_i with $1 \leq i \leq n$.*

As formalized in the following result of [7], the WS models of a program P coincide with its stable models.

Theorem 1. *Let P be a normal logic program over $\mathcal{L}$. An interpretation M over $\mathcal{L}$ is a* well-supported model of P *iff it is a stable model of P.*

2.2 Semantics for DyLPs and MDyLPs

To represent negative information in logic programs and their updates, DyLPs use *generalized logic programs* (GLPs) [16], which allow for default negation not only in the premises of rules but also in their heads. A GLP over a language $\mathcal{L}$ is any (possibly countably infinite) set of rules of the form $L_0 \leftarrow L_1, \ldots, L_n$, where each L_i is a literal of $\mathcal{L}$. Given a rule τ as above, by $hd(\tau)$ we mean L_0 and by $B(\tau)$ we mean $\{L_1, \ldots, L_n\}$.

A *dynamic logic program* $\mathcal{P}$ over a language $\mathcal{L}$ is a finite sequence $P_1, \ldots, P_m$, where all the P_i's are GLPs over $\mathcal{L}$. We call the P_is *updates*. A *multidimensional dynamic logic program* $\mathcal{MP}$ is any partially ordered finite multiset of GLPs. Let $\mathcal{M}$ be a set of indices for the elements of $\mathcal{MP}$, and let $\prec$ be the partial order defined over $\mathcal{MP}$. For any index i, by P_i we denote the element of $\mathcal{MP}$ associated with i, and we call it an *update*. We often use the notation $i \prec j$ instead of $P_i \prec P_j$. Let τ and η be two rules appearing in $\mathcal{MP}$. As an abuse of notation, we also use the notation $\tau \prec \eta$ for denoting that τ and η belong, respectively, to the updates P_i and P_j with $P_i \prec P_j$. Two rules τ and η are said to be *not comparable* iff neither $\tau \prec \eta$ nor $\eta \prec \tau$ is true. For elements P_i and P_j of $\mathcal{M}$, we say that P_i is *less recent* than P_j iff $i \prec j$. Let P_n be an update of $\mathcal{MP}$. The *genealogy of* P_n, denoted by $\mathcal{MP}^n$, is the subset of the elements of $\mathcal{MP}$ which are less recent than P_j plus P_n itself. We use $\rho(\mathcal{P})$ to denote the multiset of all rules appearing in $\mathcal{MP}$ and $\rho(\mathcal{P}^n)$ to denote the multiset of all rules appearing in $\mathcal{MP}^n$. Note that, if the order defined over the MDyLP $\mathcal{MP}$ consisting of m elements is a *total order*, then $\mathcal{MP}$ is the DyLP $P_1, \ldots, P_m$ where P_i denotes the ith element of $\mathcal{MP}$ such that $P_i \prec P_j$ iff $i < j$.

The *dynamic stable models semantics* for MDyLPs ($\mathcal{DS}$) is defined in [13] by assigning to each MDyLP a set of dynamic stable models. The basic idea of the semantics is that, if a later rule τ has a true body, then former rules in conflict with τ are *rejected*. Moreover, any atom A for which there is no rule with true body is considered false by default. The semantics is then defined by a fixpoint equation that, given an interpretation I, tests whether I has exactly the

consequences obtained after removing from the multiset $\rho(\mathcal{P}^n)$ all the rejected rules, and imposing all the default assumptions given I. Formally:

Definition 2. *Let $\mathcal{MP}$ be any MDyLP over language $\mathcal{L}$, P_n be an update of $\mathcal{MP}$ and M be a two valued interpretation. Define*

$$Default(\mathcal{MP}^n, M) = \{not\, A \,|\, \nexists\, A \leftarrow body \in \rho(\mathcal{P}^n) : body \subseteq M\}$$
$$Rej(\mathcal{MP}^n, M) = \{\tau \in P_i |\ \exists\, \eta \in P_j : i \prec j,\ \tau \bowtie \eta, B(\eta) \subseteq M\},$$

where $\tau \bowtie \eta$ means that τ and η are conflicting rules, i.e. the head of τ is the default complement of the head of η. Then M is a dynamic stable model *of $\mathcal{MP}$at P_n iff M is a fixpoint of $\Gamma^n_{\mathcal{MP}}$, defined by*

$$\Gamma^n_{\mathcal{MP}}(M) = least\,(\rho(\mathcal{P}^n) \setminus Rej(\mathcal{MP}^n, M) \cup Default(\mathcal{MP}^n, M))$$

where $least(P)$ denotes the least Herbrand model of the definite program obtained by considering each negative literal $not\, A$ in P as a new atom[2].

Other semantics for either DyLPs and MDyLPs based on causal rejection are the justified update ($\mathcal{JU}$) [15] and the update programs semantics ($\mathcal{UP}$) [6]. The latter is equivalent to the semantics proposed in [5]. All these semantics are extensions of the stable model semantics for normal and generalized LPs [8]; relations among them have been studied in [11,12], and the main result is as follows. Given any MDyLP $\mathcal{MP}$, let $\mathcal{UP}(\mathcal{P}), \mathcal{JU}(\mathcal{P}), \mathcal{DS}(\mathcal{P})$ and be the set of models of $\mathcal{P}$ according to, respectively, the update programs, the justified update, the dynamic stable for MDyLPs. Then, as showed in [12,11] the following (possibly strict) inclusions hold:

$$\mathcal{UP}(\mathcal{P}) \supseteq \mathcal{JU}(\mathcal{P}) \supseteq \mathcal{DS}(\mathcal{P}) \tag{3}$$

3 Some Considerations on the Existing Semantics

As shown in [11] and [6], all the cited semantics for MDyLPs based on causal rejection coincide on large classes of programs. The examples where these semantics differ involve cyclic dependencies among rules, similar to the one illustrated in the example 1. As stated in the introduction, all the existing semantics for *MDyLPs* (wrongly) coincide on the program of example 1, i.e. they allow the program to have a model where the commission is not declined. Note that this is a consequence of the cycle introduced by the instances of the rules 1 and 2. In fact, the MDyLP obtained by removing the mentioned instances of the rules has no model according to any of the cited semantics. Indeed these two rules that should not affect the semantics allow somehow an undesired model.

Analogous behaviours are known also in the case of DyLPs as discussed in [1], where examples are given where cycles or even tautologies generate counterintuitive behaviour. In particular, all the semantics show counterintuitive behaviour

[2] Whenever clear from the context we will omit the $\mathcal{MP}$ in any of the above defined operators.

in cases where conflicting rules appear in the same update. In [1], the authors introduce the *refined extension principle* as a principle that any semantics should satisfy in order to avoid such behaviour. Moreover, the *refined dynamic stable model semantics* [1] or simply *refined semantics* for DyLPs is defined. Such semantics obeys the refined extension principle and, in fact, it avoids the mentioned counterintuitive behaviour. The definition of the refined semantics is the same of definition 2 but replacing the MDyLP $\mathcal{MP}$ with a DyLP $\mathcal{P}$ and replacing $i \prec j$ in the rejection operator by $i \leq j$. Intuitively, whenever rules with true body and with heads, respectively, A and $notA$ appear in the same update P_i they reject each other. Moreover they reject also all the previous rules with either head A and $not\,A$, and finally they also reject the default assumption $not\,A$. Hence, unless a new rule with head A or $not\,A$ and true body appears in a more recent update, the considered program has no model because it is not possible to derive neither A nor $not\,A$. The refined semantics ($\mathcal{RS}$) for DyLPs is, among the cited semantics, the one that admits the less number of models. Hence, if we restrict to DyLPs, we can refine the inclusions 2.2 as follows

$$\mathcal{UP}(\mathcal{P}) \supseteq \mathcal{JU}(\mathcal{P}) \supseteq \mathcal{DS}(\mathcal{P}) \supseteq \mathcal{RS}(\mathcal{P}) \tag{4}$$

Unfortunately, the technique of mutual rejection of rules in the same state cannot be extended to the class of MDyLPs. In fact, if we extend directly the definition of refined model to MDyLPs, Example 1 still exhibits an undesired model. This originates from the fact that, in multidimensional case, unlike in the linear one, two rules may be *non comparable* even if they do not belong to the same update. If, instead we allow non comparable rules to reject each other, the obtained semantics allows too few models. Let us consider for instance the following program $\mathcal{MP}$ $P_1 : A$ $P_2 : not\,A$ $P_3 : A$ $P_4 : not\,A$ $P_1 \prec P_2$ $P_3 \prec P_4$ According to the proposal above, the rules in P_1 and P_4 reject each other, as well as the ones in P_2 and P_3. The default assumption is rejected as well and, as a result, we cannot derive neither A nor $not\,A$ and hence $\mathcal{MP}$ would have no model. We regard this as counterintuitive, since, according to causal rejection, the facts A are rejected and $\{not\,A\}$ should be the unique model of the program. Moreover, this last counterintuitive behavior is not captured by the refined extension principle since this principle only ensures that, whenever it is satisfied, the introduction of particular kind of rules does not allow undesired models. Here the problem is exactly the opposite: the proposed semantics would have *too few* models. We conclude that it is simply not possible to *hack* the definition of the refined semantics in order to apply it to MDyLPs nor can we refer to the refined extension principle as a tool for univocally determine a semantics. What is needed is an entirely new methodology to attack the problem. The next section introduces such a methodology.

4 Well-Supported Models for MDyLPs

As we see from equation (4), the existing semantics for DyLPs can be ordered in a sequence where each semantics is a refinement of the previous one. Going right

in the sequence, the conditions for an interpretation to be accepted as a model become stricter. It seems that research is trying to discard *bad models* from the semantics and to keep only *good models*. Is this really the case? Moreover, can we consider the sequence ended by the refined semantics, or do we need to further refine it? To answer these questions we need a formal definition of what we mean by *good model*. Recalling section 2.1, we can see an interesting historical parallelism between the evolution of semantics for DyLPs and the evolution of two-valued semantics for normal LPs, where the supported, minimal supported and stable model semantics are successive refinements. We also note similarities between examples 1 and 2, 3: in them, rules that should not play any semantic role change the behavior of some semantics by introducing more models. In the static case, this behaviour is rectified by the introduction of the concept of well-supported model. Guided by this historical perspective, we extend the notion of WS model to MDyLPs.

We first note that in the definition of well-supported models for normal LPs only the level of the positive literals in the body of a rule is considered. This happens because within normal LPs only positive literals are derived by rules, and the negative ones follow by negation by default. For DyLPs and MDyLPs, however, also negative literals can be derived by rules, since we allow negative literals in rule heads. More importantly, in the static case a rule plays a role only for deciding whether or not a given literal should be true or not. In the dynamic case, a rule is also used for *rejecting* other rules. Hence the concept of well-supportedness should be applied not only to the derivation of literals but also to the rejection of rules. More precisely, we require *well-supported rejection*: a rule can reject another rule in a previous update iff the body of the rejecting rule is true and the level of the body of the rejecting rule is less than the level of its head. The following definition formalizes this idea using level mappings.

Definition 3. *Let $\mathcal{MP}$ be any MDyLP over some language $\mathcal{L}$, let P_n be an update of $\mathcal{MP}$, ℓ be any level mapping over $\mathcal{L}$ and M be any two-valued interpretation over $\mathcal{L}$. Define the* set of rejected rules *w.r.t. ℓ by*[3]

$$Rej_\ell(\mathcal{MP}, M, n) = \{\tau \in P_i \mid \exists \eta \in P_j : i \prec j \preceq n, \tau \bowtie \eta, \\ M \models body(\eta), \ell(hd(\eta)) > \ell(B(\eta)) \ \}.$$

Given the considerations above and Definition 3, it is now easy to extend the concept of WS model to MDyLPs. An interpretation M is a WS model if it is possible to find a level mapping such that: M is a model of all the rules that are not rejected w.r.t. the given level mapping and such that, for each atom A which is true in M, a non-rejected rule with head A exists, whose body is true and has level less than the level of A. The formal definition follows.

Definition 4. *Let $\mathcal{MP}$ be any MDyLP over some language $\mathcal{L}$ and let M be an interpretation over $\mathcal{L}$, P_n be an update of $\mathcal{M}$ and M be any interpretation over $\mathcal{L}$. We say that M is a* ***well-supported model at*** *P_n iff there exists a*

[3] Hereafter, we use the simplified notation $Rej_\ell(M, n)$ whenever this causes no ambiguity.

level mapping ℓ over $\mathcal{L}$ such that i) M is a model of $\rho(P) \setminus Rej_\ell(M, n)$ and ii) $\forall\ A \in M\ \exists\ \tau \in \rho(P) \setminus Rej_\ell(M, n)$ such that $hd(\tau) = A,\ \ell(A) > \ell(B(\tau))$ and τ is supported by M.

If we consider again the program of the example 1, we find that the model where the commission x_3 is not declined is not a well supported model. This also implies that the well supported model semantics for MDyLPs does not coincide with any of the existing semantics for such class of programs. If the partial order defined over the programs is a total one, then the considered program is a DyLP. In this case, it is possible to prove the analogue of Theorem 1.

Theorem 2. *Let $\mathcal{P}$ be any DyLP and M be an interpretation. Then M is a refined stable model of $\mathcal{P}$ iff it is a well-supported model of $\mathcal{P}$.*

For this reason, we refer to the defined semantics also as to the *the refined semantics for MDyLPs*. We have shown that the WS models coincide with the refined SMs for a DyLP.

Some comparisons of the refined semantics to other semantics for MDyLPs are in order. We would expect that any well supported model of a given program is also a model in any of the existing semantics for MDyLPs. In fact this result holds, as stated by the next theorem.

Theorem 3. *Let $\mathcal{MP}$ be any MDyLPs in the language $\mathcal{L}$, P_n be an update of $\mathcal{MP}$ and M be a well supported model of $\mathcal{MP}$ at P_n. Then M is also a model of $\mathcal{MP}$ in any of the semantics defined in [13,5,11].*

It is now easy to understand the different behaviour of the considered semantics. Two distinguished semantics differ for those cases when one semantics is a better approximation of the semantics of well-supported models than the other one.

5 Fixpoint Characterization for Well-Supported Models of MDyLPs

Well-supported models define a semantics for MDyLPs. However, this characterization is purely descriptive, which is obviously not entirely satisfactory for computational purposes. Moreover, to understand from this definition whether a given interpretation is a WS model of a given MDyLP, we have to face the problem of finding a corresponding level mapping or show that such a level mapping does not exist. This does not seem to be a reasonable approach for computing the semantics and, furthermore, may not lead to quick ways of testing whether a given interpretation is a well-supported model or not. For these reasons, we present an alternative characterization based on a fixpoint operator. We characterize our models as the fixpoints of an operator defined from interpretations to interpretations, in the spirit of the Gelfond-Lifschitz operator [8]. Given a program $\mathcal{MP}$ and an update P_n with index n, we associate with any pair $(\mathcal{MP}, n)$ an operator $\Gamma^S_{(\mathcal{MP},n)}$ as in Definition 7 below. We then obtain a characterization of the well-supported models of $\mathcal{MP}$ as the fixpoints of this operator.

The semantics of a MDyLP $\mathcal{MP}$ is defined with respect to the updates P_n of $\mathcal{MP}$. To establish the semantics consider only the genealogy of P_n. The underlying idea of the semantics is to collect in a single program all the rules of the given MDyLP and add new rules and predicates that specify whether a rule is rejected or not.

Let $\mathcal{MP}$ be a MDyLP over $\mathcal{L}$, P_j, P_k be programs of $\mathcal{MP}$ and $j \prec k$. If there exists a rule γ in P_k with head L, then every rule in P_j with head $not\, L$ could be rejected, depending on whether the body of γ is true or not. Formally:

Definition 5. *Let $\mathcal{MP}$ be an MDyLP over the language $\mathcal{L}$. Add new atoms $rej(L, i)$ not belonging to $\mathcal{L}$, where L is a literal in $\mathcal{L}$ and i ranges over the indices of the updates of $\mathcal{MP}$. The set of rejecting rules over the extended language is then defined as follows:*

$$Rj(\mathcal{MP}) = \{\ rej(not\, L, j)\ \leftarrow\ body.\ \mid L \leftarrow body. \in P_k\ \wedge\ j \prec k\ \}$$

Let $\mathcal{MP}$ be a multidimensional DyLP over $\mathcal{L}$, n be an index, L be a literal of $\mathcal{L}$, M be an interpretation over $\mathcal{L}$, P_i, P_j and P_k be programs of $\mathcal{MP}^n$ and τ^i and η^j be rules in, respectively P_i and P_j. We say that η^j *is a threat for L in* i, or alternatively that L *is threatened by* η^j iff the head of η^j is $not\, L$, its body is true in M, and $j \not\prec i$. We say that η^j is a threat for some other rule τ^i in P_i iff it is a threat for its head in i. A literal (rule) is considered to be *strictly safe* in P_i iff all its threats are rejected. Intuitively, derivations should only be made from safe rules. The main idea behind our definition is that a rule can be used to derive consequences iff it has already been established that the rule is safe. To achieve this result, we first consider the GLP given by the union of all the rules in $\mathcal{MP}$ with a new atom in the body of each rule which is satisfied only if the considered rule is safe. Then we introduce rules specifying which threats should be rejected in order to consider a literal (rule) as safe. Finally we introduce rules determining when a threat is rejected. Formally:

Definition 6. *Let $\mathcal{MP}$ be any multidimensional dynamic logic program in the language $\mathcal{L}$. Add new atoms $safe(L, i)$ not belonging to $\mathcal{L}$, where L is a literal in $\mathcal{L}$ and i ranges over the indices of the updates of $\mathcal{MP}$. We denote by $\Sigma(\mathcal{MP})$ the following set of rules.*

$$\Sigma(\mathcal{MP}) = \{L \leftarrow body, safe(L, i) | L \leftarrow body \in P_i\}$$

Let M be an interpretation over $\mathcal{L}$. The set of conditions for a literal L to be strictly safe at P_i is defined as follows.

$$cond^S(\mathcal{MP}, M, L, i) = \{rej(not\, L, j) \mid \exists \eta \in P_j, j \not\prec i,\ M \models B(\eta) \wedge hd(\eta) = not\, L\}$$

By abuse of notation, $cond^S(\mathcal{MP}, M, L, i)$ also denotes the conjunction of all the literals in $cond^S(\mathcal{MP}, M, L, i)$. The set of strictly safe rules *is defined as*

$$Safe^S(\mathcal{MP}, M)) = \{safe(L, i) \leftarrow cond^S(\mathcal{MP}, M, L, i) | \exists\ \tau \in P_i :\ hd(\tau) = L\}.$$

Having specified the condition for a rule to be allowed to derive literals, we then add the set of default assumptions and compute the least model of the obtained program. Finally we discard the auxiliary literals computed.

Definition 7. *Let $\mathcal{MP}$ be any multidimensional dynamic logic program in the language $\mathcal{L}$, P_n be an update of $\mathcal{MP}$ and M be an interpretation over $\mathcal{L}$. We define the operator $\Gamma^S_{(\mathcal{MP},n)}$ on interpretations over $\mathcal{L}$ as follows.*

$$\Gamma^S_{(\mathcal{MP},n)}(I) = least\ (\Sigma(\mathcal{MP}^n) \cup Safe^S(\mathcal{MP}^n, I) \cup Rj(\mathcal{MP}^n) \cup Default(\mathcal{MP}^n, I)\ )|_{\mathcal{L}}$$

We are finally ready to define the refined semantics for MDyLPs. A *refined multi stable model* (RMSM) of an MDyLP at P_n is any fixpoint of the $\Gamma^S_{(\mathcal{MP},n)}$ operator.

Definition 8. *Let $\mathcal{MP}$ be any multidimensional dynamic logic program in the language $\mathcal{L}$, P_n be an element of $\mathcal{MP}$ and M be any interpretation over $\mathcal{L}$. We say that M is a* refined multi stable model *of $\mathcal{MP}$ at P_n iff $M = \Gamma^S_{(\mathcal{MP},n)}(M)$.*

The goal of Definition 8 is to provide a fixpoint characterization of the WS models of Definition 4. Indeed, this has been accomplished, as shown by the following theorem.

Theorem 4. *Let $\mathcal{MP}$ be any MDyLP in the language $\mathcal{L}$, n be an index and M be any two valued interpretation over $\mathcal{L}$. Then M is a* refined multi stable model *of $\mathcal{MP}$ at P_n iff M is a* well-supported model *of $\mathcal{MP}$ at P_n.*

6 Conclusions and Future Research

The initial purpose of the paper was to provide a semantics for MDyLPs based on causal rejection that can be properly considered to be *the* stable models-like semantics for such classes of programs. To obtain this, we extended the definition of well-supported model to the dynamic case. It turns out that, for DyLPs, our characterization coincides with the refined semantics. We provided also a fixpoint characterization of such semantics and established relationships between the new semantics and existing ones. Is it possible to conclude that the refined semantics is the proper extension of the SMs semantics to DyLPs and MDyLPs? Unfortunately there exists no theoretical result which is equivalent to the statement "*this is the correct semantics*". Nevertheless we claim that the characterizations given herein provide some evidence that further refinements of the semantics will not be necessary.

As mentioned in the introduction, there exist stable models-like semantics for LP updates [17,19,18] which are not (or at least not exclusively) based on the concept of causal rejection. Unlike those based on causal rejection, such semantics significantly differ from the refined semantics, and among each other, in properties, behaviors and underlying concepts. Comparisons of the semantics defined in [17,19] can be found in [12]. Given the underlying differences, it is possible that in the future specific application areas will be found where different

approaches to LP updates are needed for different applications. It is our strong opinion that DyLPs and MDyLPs can be a useful tool in several application areas, in particular in those areas related to web-oriented applications for AI where powerful reasoning capabilities have to be applied in highly dynamic environments and where merging knowledge from different sources is an important and challenging task.

References

1. J. J. Alferes, F. Banti, A. Brogi, and J. A. Leite. Semantics for dynamic logic programming: a principled based approach. In *LPNMR-7*. Springer, 2004.
2. J. J. Alferes, J. A. Leite, L. M. Pereira, H. Przymusinska, and T. C. Przymusinski. Dynamic updates of non-monotonic knowledge bases. *The Journal of Logic Programming*, (1–3):43–70, 2000.
3. J.J. Alferes, F. Banti, and A. Brogi. From logic programs updates to action description updates. In J. Leite and P. Torroni, editors, *CLIMA V*, pages 227–242. Pre-Proceedings, 2004. ISBN: 972-9119-37-6.
4. K. R. Apt and R. N. Bol. Logic programming and negation: A survey. *The Journal of Logic Programming*, 19–20:9–72, 1994.
5. F. Buccafurri, W. Faber, and N. Leone. Disjunctive logic programs with inheritance. In D. De Schreye, editor, *ICLP-99*, 1999.
6. T. Eiter, M. Fink, G. Sabbatini, and H. Tompits. On properties of semantics based on causal rejection. *Theory and Practice of Logic Programming*, 2:711–767, 2002.
7. François Fages. Consistency of Clark's completion and existence of stable models. *Journal of Methods of Logic in Computer Science*, 1:51–60, 1994.
8. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. Kowalski and K. A. Bowen, editors, *ICLP-5*, 1988.
9. P. Hitzler. Towards a systematic account of different logic programming semantics. In A. Günter, R. Kruse, and B. Neumann, editors, *KI2003*, 2003.
10. P. Hitzler and M. Wendt. A uniform approach to logic programming semantics. *Theory and Practice of Logic Programming*, 5(1–2):123–159, 2005.
11. M. Homola. Dynamic logic programming: Various semantics are equal on acyclic programs. In J. Leite and P. Torroni, editors, *CLIMA V*, pages 227–242. Pre-Proceedings, 2004. ISBN: 972-9119-37-6.
12. J. A. Leite. *Evolving Knowledge Bases*, volume 81 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2003.
13. J. A. Leite, J. J. Alferes, and L. M. Pereira. Multi-dimensional dynamic knowledge representation. In T. Eiter, M. Truszczynski, and W. Faber, editors, *LPNMR-01*, pages 365–378, 2001.
14. J. A. Leite, J. J. Alferes, and L. M. Pereira. Minerva – a dynamic logic programming agent architecture. In *Intelligent Agents VIII*, volume 2333 of *LNAI*. 2002.
15. J. A. Leite and L. M. Pereira. Iterated logic program updates. In *JICSLP-98*, 1998.
16. V. Lifschitz and T. Woo. Answer sets in general non-monotonic reasoning. In B. Nebel, C. Rich, and W. Swartout, editors, *KR-92*, 1992.
17. C. Sakama and K. Inoue. Updating extended logic programs through abduction. In M. Gelfond, N. Leone, and G. Pfeifer, editors, *LPNMR-99*, 1999.
18. J. Sefranek. A Kripkean semantics for dynamic logic programming. In *LPAR'2000*. Springer, 2000.
19. Y. Zhang and N. Y. Foo. Updating logic programs. In Henri Prade, editor, *ECAI-98*, 1998.

Application of Smodels in Quartet Based Phylogeny Construction

Gang Wu, Jia-Huai You, and Guohui Lin

Department of Computing Science, University of Alberta, Edmonton, Alberta T6G 2E8, Canada

Abstract. Evolution is an important sub-area of study in biological science, where given a set of taxa, the goal is to reconstruct their evolutionary history, or phylogeny. One very recent approach is to predict a local phylogeny for every subset of 4 taxa, called a quartet topology, and then to assemble a phylogeny for the whole set of taxa satisfying these predicted quartet topologies. In general, the predicted quartet topologies might not always agree with each other, and thus the objective function becomes to satisfy a maximum number of them. This is the well known Maximum Quartet Consistency (MQC) problem. In the past, the MQC problem has been solved by dynamic programming and the so-called fixed-parameter method. Recently, we have proposed to solve the MQC in answer set programming. In this note, we summarize the theoretical results of this approach and report new experimental results for the purpose of comparison, which show that our approach in answer set programming is favored over the existing approaches based on dynamic programming and fixed-parameter method. In particular, some of the hard instances (where the error ratio is high) that were not reported to be solved in other approaches can now be solved in our approach.

1 Introduction

In recent years, quartet based phylogeny construction methods have received considerable attention in the computational biology community. In comparison with other phylogeny construction methods, an advantage of quartet based methods is that they can overcome the data disparity problem [1]. Given a taxa set S, quartet based phylogeny construction methods first try to build an unrooted phylogeny for every (or most) subset of four taxa. This four taxa set is called a *quartet* and the phylogeny associated with that quartet is called a *quartet topology*. For each quartet, there are 3 possible quartet topologies associated with it. For example, Fig. 1 shows the 3 quartet topologies for quartet $\{s_1, s_2, s_3, s_4\}$. For simplicity, we use $[s_1, s_2|s_3, s_4]$ to denote the quartet topology in which the path connecting s_1 and s_2 doesn't intersect the path connecting s_3 and s_4 (cf. Fig. 1(a)). Given a phylogeny T on a set of taxa S and a quartet topology q on $\{s_i, s_j, s_k, s_\ell\}$, if the path structure connecting s_i, s_j, s_k, and s_ℓ in T is the same as q, then T *satisfies* q or q is *consistent* with T. Given a set Q of quartet topologies, the recognition problem, called the *Quartet Compatibility Problem* (QCP), is to determine if there is a phylogeny T on S satisfying all the quartet topologies in Q. The more

C. Baral et al. (Eds.): LPNMR 2005, LNAI 3662, pp. 369–373, 2005.
© Springer-Verlag Berlin Heidelberg 2005

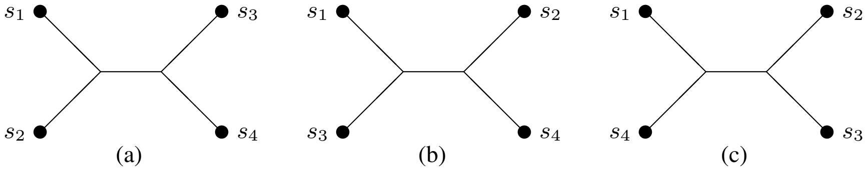


Fig. 1. Three possible quartet topologies for quartet $\{s_1, s_2, s_3, s_4\}$: (a) $[s_1, s_2|s_3, s_4]$, (b) $[s_1, s_3|s_2, s_4]$, and (c) $[s_1, s_4|s_2, s_3]$

interesting computational problem is the optimization problem where we cannot find a phylogeny to satisfy all the quartet topologies in Q and the goal is to construct one to satisfy as many quartet topologies as possible. This is the so-called *Maximum Quartet Consistency Problem* (MQC), and it is NP-hard [2].

A few attempts have been made to solve the MQC problem optimally. Ben-Dor et al. [1] present a dynamic programming approach whose time complexity is $O(3^n n^4)$, where $n = |S|$ is the number of taxa. The fixed-parameter method proposed by Gramm and Niedermeier [3] can return an optimal phylogeny in $O(4^k n + n^4)$ time, where k is the *exact* number of of inconsistent quartet topologies.

2 Solving the MQC Problem with Answer Set Programming

We consider the rooted phylogeny, as the quartet consistency property is independent of whether or not a phylogeny is rooted. Given a set of taxa $S = \{s_1, s_2, \ldots, s_n\}$ and a rooted phylogeny T on S, the *least common ancestor* of two leaf nodes s_i and s_j in T is the common ancestor of s_i and s_j farthest away from the root, denoted as LCA(s_i, s_j). A *labeling scheme* for T is a mapping from the set of internal nodes in T to the set of integers $\{1, 2, \ldots, n-1\}$. Note that there are exactly $(n-1)$ internal nodes in T and each node can be labeled by any number in the set $\{1, 2, \ldots, n-1\}$. A labeling scheme is *ultrametric* if along any root to leaf path, the labels of the internal nodes on the path are strictly decreasing. One rooted phylogeny together with an ultrametric labeling scheme is called an *ultrametric phylogeny*. Let M be an $n \times n$ symmetric matrix in which $M(i, j)$ denotes the label of the internal node LCA(s_i, s_j). A triplet (i, j, k) for M is *ultrametric* if and only if there are two equal values among $M(i, j)$, $M(j, k)$, and $M(i, k)$ and they are greater than the third value. M is *ultrametric* if every triplet (i, j, k) for M is ultrametric. An $n \times n$ ultrametric matrix M *satisfies* a quartet topology $[s_i, s_j|s_k, s_\ell]$ if $\min\{M(i, k), M(j, \ell)\} > \min\{M(i, j), M(k, l)\}$ holds.

Theorem 1. [5] *Given a quartet topology set Q and an ultrametric phylogeny T on S, T satisfies a maximum number of quartet topologies in Q if and only if the corresponding ultrametric matrix M satisfies a maximum number of quartet topologies in Q.*

Theorem 1 tells that the MQC problem is equivalent to the search for an ultrametric matrix to satisfy a maximum number of quartet topologies. In such a way, the MQC problem can be formulated into a constraint programming problem as shown in Fig. 2. This formulation can be easily translated into answer set programming rules and implemented in Smodels [4]. For each variable $M(i, j)$ with its domain $\{1, 2, \ldots, n-1\}$, we use one atom $m(i, j, k)$ to represent that $M(i, j)$ takes the value k. That is, when $m(i, j, k)$ is true, $M(i, j) = k$. We know that each variable should take exactly one value in the domain. So we use a *cardinality rule* to represent this constraint.

INPUT:

- A set of n taxa $S = \{s_1, s_2, \ldots, s_n\}$. n^2 variables $M(i,j)$, where $1 \leq i,j \leq n$. The domain of each variable $M(i,j)$, where $i \neq j$, is $\{1, 2, \ldots, n-1\}$; $M(i,i) = 0$, for all i.
- A set Q of quartet topologies on S. Each quartet topology $[s_i, s_j | s_k, s_\ell]$ is transformed into a quartet consistency constraint $q(i,j,k,\ell)$.

CONSTRAINTS:

- Symmetry Constraint: $M(i,j) = M(j,i)$, for all $1 \leq i,j \leq n$, i,j distinct;
- Ultrametric Constraint: $ultra(i,j,k)$, for all $1 \leq i,j,k \leq n$, i,j,k distinct;
- Quartet Consistency Constraint: $q(i,j,k,\ell)$, for quartet topology $[s_i, s_j | s_k, s_\ell] \in Q$.

GOAL:

- Find a solution to $M(i,j)$ for all $1 \leq i,j \leq n$, such that all the symmetry and ultrametric constraints are satisfied and the number of satisfied quartet consistency constraints is maximized.

Fig. 2. Formulating the MQC problem into a constraint programming problem

For example, "$1\{m(1,2,1), m(1,2,2), m(1,2,3), m(1,2,4)\}1$" means that variable $M(1,2)$ takes exactly one value in the domain $\{1,2,3,4\}$. Similarly, we use one atom $ultra(i,j,k)$ to represent that the triplet (i,j,k) for M is ultrametric if $ultra(i,j,k)$ is true, and one atom $q(i,j,k,\ell)$ to represent that $[s_i, s_j | s_k, s_\ell]$ is in the optimal phylogeny if $q(i,j,k,\ell)$ is true. Based on this translation of variables, the ultrametric constraints and quartet consistency constraints can be formulated as:

$$\begin{aligned} ultra(i,j,k) &\leftarrow\ equal(i,j,j,k),\ gter(i,j,i,k) \\ ultra(i,j,k) &\leftarrow\ equal(i,j,i,k),\ gter(i,j,j,k) \\ ultra(i,j,k) &\leftarrow\ equal(i,k,j,k),\ gter(i,k,i,j) \\ q(i,j,k,\ell) &\leftarrow\ gter(i,k,i,j),\ gter(j,\ell,i,j) \\ q(i,j,k,\ell) &\leftarrow\ gter(i,k,k,\ell),\ gter(j,\ell,k,l) \end{aligned}$$

where $equal(i,j,j,k)$ means $M(i,j) = M(j,k)$, and $gter(i,j,j,k)$ means $M(i,j) > M(j,k)$. Finally, we use a $maximize$ statement to find a stable model that contains the maximum number of true assignments of $q(i,j,k,\ell)$. For example, for $n = 5$,

$$\begin{aligned} maximize\ [\ & q(2,3,4,5) = 1, q(1,4,3,5) = 1, q(1,5,2,4) = 1, \\ & q(1,5,2,3) = 1, q(1,4,2,3) = 1] \end{aligned}$$

3 Strategies to Speed Up the Computation

Observe that an ultrametric matrix M is symmetric and therefore instead of putting the symmetry as constraints, we would rather use it to reduce the number of variables. Only $M(i,j)$ with $1 \leq i < j \leq n$ becomes a variable, which gives only $\frac{1}{2}(n^2 - n)$ variables at the end. Consequently, we remove all symmetry constraints from the constraint set. Similarly, we would only consider ultrametric constraints $ultra(i,j,k)$ such that $1 \leq i < j < k \leq n$ and quartet consistency constraints $q(i,j,k,\ell)$ such that $1 \leq i < j \leq n$, $1 \leq k < \ell \leq n$, and $1 \leq i < k \leq n$.

In the following, we introduce two new speedup strategies used in our Smodel programs. For any rooted phylogeny T, one natural ultrametric labeling scheme is to label

each internal node by its height. Therefore, the domain of variables $M(i, j)$ can be reduced to $\{1, 2, \ldots, h\}$, where h denotes the height of the optimal phylogeny.

Theorem 2. *Given a quartet topology set Q on taxa set $S = \{s_1, s_2, \ldots, s_n\}$, there exists an optimal phylogeny whose height is at most $\lceil \frac{n}{2} \rceil$.*

Theorem 3 lists some of the propagation rules that have been implemented into our Smodels programs.

Theorem 3. *For every set of five taxa $\{s_i, s_j, s_k, s_\ell, s_m\}$,*

- *if $[s_i, s_j|s_k, s_\ell]$ and $[s_i, s_j|s_k, s_m]$ are in an optimal phylogeny, then $[s_i, s_j|s_\ell, s_m]$ must be in the optimal phylogeny too, i.e.,*

$$q(i, j, \ell, m) \leftarrow q(i, j, k, \ell),\ q(i, j, k, m)$$

- *if $[s_i, s_j|s_k, s_\ell]$ and $[s_i, s_k|s_\ell, s_m]$ are in an optimal phylogeny, then $[s_i, s_j|s_k, s_m]$, $[s_i, s_j|s_\ell, s_m]$, and $[s_j, s_k|s_\ell, s_m]$ must be in the optimal phylogeny too, i.e.,*

$$\begin{aligned} q(i, j, k, m) &\leftarrow q(i, j, k, \ell),\ q(i, k, \ell, m) \\ q(i, j, \ell, m) &\leftarrow q(i, j, k, \ell),\ q(i, k, \ell, m) \\ q(j, k, \ell, m) &\leftarrow q(i, j, k, \ell),\ q(i, k, \ell, m). \end{aligned}$$

4 Computational Results

We have shown in [5] some preliminary computational results on our answer set programming approach. In this note, we make comparisons among the approaches proposed to solve the MQC problem optimally, including the dynamic programming by Ben-Dor et al. (denoted as DP) [1], the fixed-parameter method (denoted as GN) [3], and our answer set programming (denoted as ASP). We generated datasets defined by a pair (n, p) where for each dataset, n is the number of taxa and p records the percentage of quartet errors in the given complete quartet set. We used quartet error percentage $p = 1\%, 5\%, 10\%, 15\%, 20\%, 30\%$. The reported experimental results were done on a computer with an AMD 2.2GHz Opteron processor and 2.5GB main memory. Some of the running time results are summarized in Table 1. These results showed that the quartet error rate may affect the performance greatly, typically when the number of taxa under consideration exceeds 15. When the error rate was less than 10%, the problem could be solved more efficiently.

In our experiments, we found that the *lookahead* function employed by Smodels played a very important role during the search of an optimal ultrametric matrix. In particular, we solved the Smodels programs without lookahead on the same datasets of 10 and 15 taxa reported in Table 1. The average computational times were 2 minutes and 5 minutes, respectively. These results showed that the lookahead in Smodels made hundreds of speedups to our Smodels programs and was the main factor that makes our programs outperform the other approaches.

The symmetry breaking and domain reduction strategies proposed in Section 3 obviously can reduce the running time significantly since they can eliminate many variables and reduce the domain for variables. In our experiments, Smodels could improve search efficiency with the propagation rules proposed in Section 3. However, the overhead is a concern since Smodels need extra time to compute these rules. In some of our datasets, the performance with all possible propagation rules was even worse than that without

Table 1. The running times of three exact methods for the MQC problem. A '–' indicates that a method didn't terminate in 168 hours (7 days).

Problem Size		DP	GN	ASP	Problem Size		DP	GN	ASP
$n = 10$	$p = 1\%$	2 secs	1 secs	1 sec	$n = 15$	$p = 1\%$	2 mins	20 secs	1 sec
	$p = 5\%$	2 secs	1 secs	1 sec		$p = 5\%$	2 mins	10 mins	1 sec
	$p = 10\%$	2 secs	5 secs	1 sec		$p = 10\%$	2 mins	5 hours	1 sec
	$p = 15\%$	2 secs	16 secs	1 sec		$p = 15\%$	2 mins	–	1 sec
	$p = 20\%$	2 secs	35 secs	1 sec		$p = 20\%$	2 mins	–	1 sec
	$p = 30\%$	2 secs	2 mins	1 sec		$p = 30\%$	2 mins	–	1 sec
$n = 20$	$p = 1\%$	40 hrs	20 mins	10 mins	$n = 25$	$p = 1\%$	–	9 hrs	20 mins
	$p = 5\%$	40 hrs	–	40 mins		$p = 5\%$	–	–	8 hrs
	$p = 10\%$	40 hrs	–	6 hrs		$p = 10\%$	–	–	78 hrs
	$p = 15\%$	40 hrs	–	6 hrs		$p = 15\%$	–	–	102 hrs
	$p = 20\%$	40 hrs	–	8 hrs		$p = 20\%$	–	–	136 hrs
	$p = 30\%$	40 hrs	–	10 hrs		$p = 30\%$	–	–	–

any propagation rules. Currently, we are investigating this issue and trying to figure out which types of propagation rules are more important than the others.

5 Conclusions

We have presented a formulation of the MQC problem and applied Smodels to solve it. The formulation, together with our speedup strategies, might lead us to a new perspective of the problem, as our experiments on simulated datasets showed that the proposed approach outperformed existing approaches proposed to solve the MQC problem optimally. Although in the worst case solving the problem still takes exponential time, it allows us to incorporate the domain knowledge into the search process. In the ideal case, we might be able to encode the target matrix variables such that the exponential behavior is a rare occurrence, and the average behavior is acceptable for practical use.

References

1. A. Ben-Dor, B. Chor, D. Graur, R. Ophir, and D. Pelleg. From four-taxon trees to phylogenies: the case of mammalian evolution. In *Proceedings of the Fourth Annual International Computing and Combinatorics Conference (RECOMB)*, pages 9–19, 1998.
2. V. Berry, T. Jiang, P. E. Kearney, M. Li, and H. T. Wareham. Quartet cleaning: Improved algorithms and simulations. In *Proceedings of the 7th Annual European Symposium on Algorithms (ESA'99)*, LNCS 1643, pages 313–324, 1999.
3. J. Gramm and R. Niedermeier. A fixed-parameter algorithm for minimum quartet inconsistency. *Journal of Computer and System Science*, 67:723–741, 2003.
4. P. Simons. *Smodels: an implementation of the stable model semantics for logic programs*. Accessible through `http://www.tcs.hut.fi/Software/smodels/`.
5. G. Wu, G.-H. Lin, and J. You. Quartet based phylogeny reconstruction with answer set programming. In *Proceedings of The 16th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2004)*, pages 612–619, 2004.

Using Answer Set Programming for a Decision Support System

Christoph Beierle[1], Oliver Dusso[1], and Gabriele Kern-Isberner[2]

[1] Dept. of Computer Science, FernUniversität in Hagen, 58084 Hagen, Germany
[2] Dept. of Computer Science, University of Dortmund, 44221 Dortmund, Germany

Abstract. ACMI is a decision support system for the checking of medical invoices in a German health insurance company. We present a brief overview of the system and its implementation in DLV.

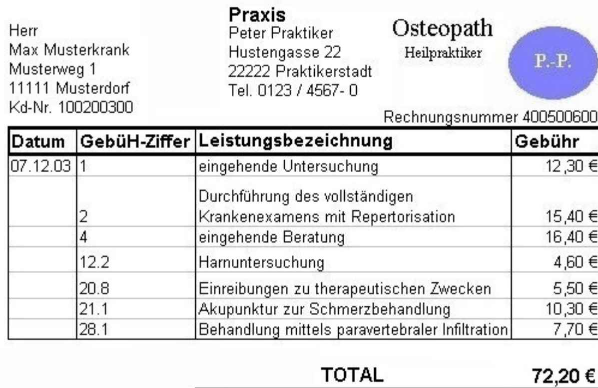
1 Introduction

In contrast to Germany's compulsory health insurance, in the private health insurance system, a physician or a physiotherapist does not have a contractual relationship with the patient's insurance company, but only with the patient himself. He issues an invoice to the patient which the patient has to pay. Having a private health insurance, the patient will hand the doctor's bill to his insurance company for reimbursement. Typically, the insurance company will check whether the invoice obeys various legal and other regulations, in particular, whether it conforms to e.g. the *Gebührenordnung für Ärzte* (GOÄ, scale of fees for physicians) [4]. The checking of the invoices requires detailed knowledge of many regulations and is a labor-intensive task; the purpose of ACMI (Automatic Checking of Medical Invoices) is to support this task [2]. When investigating the official regulations concerning medical invoices (e.g. [4], [5]) and the corresponding business rules of one of Germany's large insurance companies, it turned out that also various default rules are used. Thus, the paradigm of answer set programming (ASP) [6] was used for ACMI's implementation.

2 Overview of the System

ACMI is embedded in an environment processing the workflow from the incoming patients invoices to the refunding decision done by the insurance company's person in charge. Figure 1 shows a typical invoice. The first column specifies the date when the treatment was carried out. The second column specifies a number from the respective *Gebührenordnung* (scale of fees). In this case, the invoice is from an alternative practitioner (*Heilpraktiker*) so the relevant scale of fees is the GebüH (*Gebührenverzeichnis für Heilpraktiker*, scale of fees for alternative practitioners) [5]. The third column gives a verbal description of the treatment corresponding to the fee number, and the fourth column contains the invoiced fee (in €) for that treatment (which is not fixed, but must be in a certain range

C. Baral et al. (Eds.): LPNMR 2005, LNAI 3662, pp. 374–378, 2005.
© Springer-Verlag Berlin Heidelberg 2005


Herr
Max Musterkrank
Musterweg 1
11111 Musterdorf
Kd-Nr. 100200300

Praxis
Peter Praktiker
Hustengasse 22
22222 Praktikerstadt
Tel. 0123 / 4567- 0

Osteopath
Heilpraktiker
P.-P.

Rechnungsnummer 400500600

Datum	GebüH-Ziffer	Leistungsbezeichnung	Gebühr
07.12.03	1	eingehende Untersuchung	12,30 €
	2	Durchführung des vollständigen Krankenexamens mit Repertorisation	15,40 €
	4	eingehende Beratung	16,40 €
	12.2	Harnuntersuchung	4,60 €
	20.8	Einreibungen zu therapeutischen Zwecken	5,50 €
	21.1	Akupunktur zur Schmerzbehandlung	10,30 €
	28.1	Behandlung mittels paravertebraler Infiltration	7,70 €

TOTAL 72,20 €

Fig. 1. An invoice handed in by the patient Max Musterkrank

specified by the GebüH). After extraction of the relevant information from the invoice, ACMI automatically performs checks for every item with a fee number and reports the results.

3 ACMI's Knowledge Base

In the following, we focus on ACMI's knowledge base with respect to the GebüH [5]. The GebüH contains 148 different fee numbers (as a matter of fact, the GOÄ [4] contains many more). Each combination of fee numbers occurring in an invoice for a particular day of treatment must satisfy various constraints given in the GebüH, e.g.:

C1 *Number 4 may not occur more than once, and it may occur simultaneously only with 1 or with 17.1.*
C3 *If number 28.1 occurs more than once, replace all its occurrences with one occurrence of 28.2.*
C4 *Number 12.9 may not occur simultaneously with 12.10 and 12.11.*
C5 *Number 5.0 may not occur simultaneously with 6.0, 7.0 or 8.0.*

Note that there are constraints about the frequency of certain numbers, different restrictions about the simultaneous occurrences of combinations of numbers, and even prescriptions where numbers have to be replaced by another one.

There are also internal business rules (cf. [1]) of the insurance company attached to a fee number, e.g:

T1 *Number 21.1 is only refundable in insurance contracts with full coverage of the GebüH.*
H1 *For number 12.2 a notification text is to be displayed to the person in charge.*

In addition, the following general rule applies to each fee number N:

G1 *If a number N occurs in an invoice and N is not found to be invalid, then N is valid.*

Note that this rule is a typical default rule, and it turned out that also many other rules needed (default or classical) negation and could be expressed quite straightforwardly in ASP.

4 Knowledge Representation with Rule Schemas

By investigating all individual rules, three general types of rule schemas and four different types of consequences of a rule violation could be identified. Using these observations, each of ACMI's rule information can be expressed using the following three predicates, where N is the number to be checked, $k \geq 0$ is a number indicating a maximal frequency, $\mathcal{M}$ is a set of fee numbers, $Cons \in \{reject, replace, tariff, note\}$ indicates the type of consequence of a violation, and R is a fee number being used in the output part of the rule (e.g. the number to be rejected or the number be used in a replacement):

$frequency(N, k, Cons, R)$**:** If N occurs more than k times, execute $Cons$ for N with number R.

$forbidden(N, k, \mathcal{M}, Cons, R)$**:** If N occurs and more than k fee numbers from $\mathcal{M}$ occur, execute $Cons$ for N with number R.

$restricted(N, \mathcal{M}, Cons, R)$**:** If N occurs and another fee number occurs that is not in $\mathcal{M}$, execute $Cons$ for N with number R.

In all three predicates, "execute $Cons$ for N with number R" is given by:

- If $Cons = reject$, mark N to be rejected.
- If $Cons = replace$, mark N to be rejected and mark R to be refunded instead.
- If $Cons = tariff$, mark N to be not covered by the insurance tariff unless the current insurance tariff covers the full GebüH.
- If $Cons = note$, mark N to be displayed with notification R to the person in charge.

E.g., using these predicates, the information of the rules C1, C3, C4, C5, and T1 given in Sec. 3 can be expressed by:

C1: *frequency(4, 1, replace, 4)*
restricted(4, {1, 17.1}, reject, 4)
C3: *frequency(28.1, 1, replace, 28.2)*
C4: *forbidden(12.9, 1, {12.10,12.11}, reject, 12.9)*
C5: *forbidden(5, 0, {6,7,8}, reject, 5)*
T1: *frequency(21.1, 0, tariff, 21.1)*

For instance, *frequency(4, 1, replace, 4)* says that if number 4 occurs more than once, all occurrences of 4 should be replaced by a single occurrence of 4. *restricted(4, {1, 17.1}, reject, 4)* says that 4 should be rejected if fee numbers other than 1 or 17.1 occur simultaneously with it. The schema *frequency(28.1, 1, replace, 28.2)* requires that all occurrences of 28.1 should be replaced by 28.2 if 28.1 occurs more than once. In the schema for T1, *frequency(21.1, 0, tariff, 21.1)* states that 21.1 might not be covered, depending on the current tariff. Note also the difference between the schemas for C4 and C5: 12.9 may occur with (not more than) one of {12.10, 12.11}, but 5 may not occur simultaneously with any of {6, 7, 8}.

5 Implementation of ACMI in DLV

Whereas the rule schemas given above represent a high level of abstraction, they can not be expressed directly within the paradigm of ASP due to their usage of set-valued arguments. Therefore, we developed a transformation from the general rule schemas to DLV [3] code where each schema instance corresponds to a set of DLV facts and rules.

Internally, each fee number is represented by a four digit number, e.g., 4 by 0400, 10 by 1000, and 20.8 by 2008. The occurrence of a number N at the I'th position in the invoice is given by the literal $p(I, N)$. Thus, the invoice depicted in Figure 1 is internally represented by

```
p(1,0100). p(2,0200). p(3,0400). p(4,1202).
p(5,2008). p(6,2101). p(7,2801). tariff(vc).
```

where the literal *tariff(vc)* reflects the patient's insurance tariff, being extracted from the corresponding database.

A literal of the form $z(N)$ (resp. $-z(N)$) indicates that N occurs in the invoice and should be refunded (resp. should not be refunded). A literal $r(N)$ says that N serves as a replacement for (one or more) other number occurrences. A literal $h(N)$ indicates that a notification text should be presented to the person in charge, and $t(N)$ says that the patient's tariff does not cover fee number N. Rules like

```
fullGebuh :- tariff(an).
fullGebuh :- tariff(ebc).
```

specify the tariffs providing full coverage of the GebüH. Also rule G1 (cf. Sec. 3) can be expressed easily using the DLV default rule:

```
z(N) :- p(_,N), not -z(N).
```

For each of the three rule schemas introduced in Sec. 4, one or more facts are generated to represent what is called the schema information. For each $\mathit{frequency}(N, k, \mathit{Cons}, R)$, `frequency _fact(N,k,Cons,R).` is generated. For each $\mathit{forbidden}(N, k, \mathcal{M}, \mathit{Cons}, R)$ with $\mathcal{M} = \{M1, \ldots, Mn\}$, n facts of the form

```
forbidden_fact(N, k, Mi, Cons, R).
```

are generated, and similarly for all *restricted* rule schemas.[1] For instance, for the rule schemas for C1, the following facts are generated:

```
frequency_fact(0400, 1, replace, 0400).        %
restricted_fact(0400, 0100, reject, 0400).     %   C1
restricted_fact(0400, 1701, reject, 0400).     %
```

Checking the constraints expressed by the rule schemas is done by exploiting DLV's count facility as e.g. in checking the *restricted* schema:

[1] If there is more than one rule schema of the form $\mathit{forbidden}(N, _, _, _, _)$ for a fee number N, a more complicated generation of facts along with a correspondingly extended form of checking the constraints expressed by these facts must be used [2].

```
execute(N,Cons,R) :-
  restricted_fact(N,_,Cons,R),
  countRestrictedPartners(N,I),
  countPartners(N,All),
  All > I.

countRestrictedPartners(N,I) :-
  p(_,N),
  #count{P: restrictedPartner(N,P)} = I.

restrictedPartner(N,Partner) :-
  p(_,N), p(_,Partner),
  restricted_fact(N,Partner,_,_).

countPartners(N,All) :-
  p(_,N),
  #count{P: p(_,P), P <> N} = All.
```

Execution of the consequences of a rule violation is ensured by rules like:

```
%------ replace: ---------------
-z(N) :- execute(N, replace, R).
r(R)  :- execute(N, replace, R).

%  ------ tariff: ------------
t(N) :- execute(N, tariff, R),
        not fullGebuh.
```

6 Conclusions and Future Work

Using appropriate input masks, instances of the *frequency*, *forbidden* and *restricted* schemas can be created and modified directly by the insurance experts who need not have knowledge about ASP. Additionally, the three rule schemas provide a powerful modelling tool: All rules and regulations stemming from the GebüH [5] or from the insurance company's business rules could be expressed straightforwardly. Whereas these set-valued constraints can not be expressed directly as ASP rules, we developed a transformation for generating DLV code ensuring the constraints. Up to now the implemented system covers the full GebüH [5]; our current work includes the testing of ACMI in a large insurance company and its extension to cover the full GOÄ [4].

References

1. P. A. Bonatti, N. Shahmehri, C. Duma, D. Olmedilla, W. Nejdl, M. Baldoni, C. Baroglio, A. Martelli, V. Patti, P. Coraggio, G. Antoniou, J. Peer, and N. E. Fuchs. Rule-based policy specification: State of the art and future work. Project Deliverable D1, Working Group I2, EU NoE REWERSE, September 2004.
2. O. Dusso. Entscheidungssysteme für die Rechnungprüfung in der Krankenversicherung mit logik-basierten Regelsprachen. Diplomarbeit, Fachbereich Informatik, FernUniversität Hagen, 2005.
3. T. Eiter, W. Faber, N. Leone, and G. Pfeifer. Declarative problem solving using the DLV system. In J. Minker, editor, *Logic-Based Artificial Intelligence*, pages 79–103. Kluwer Academic Publishers, Dordrecht, 2000.
4. GOÄ – Gebührenordnung für Ärzte. `www.e-bis.de/goae/defaultFrame.htm`, 2004.
5. GebüH – Gebührenverzeichnis für Heilpraktiker. `www.patientenprojekt.de/Gebuehrenverzeichnis%20Heilpraktiker.pdf`, 2002.
6. M. Gelfond and N. Leone. Logic programming and knowledge representation – the A-Prolog perspective. *Artificial Intelligence*, 138:3–38, 2002.

Data Integration: A Challenging ASP Application

Nicola Leone[1], Thomas Eiter[2], Wolfgang Faber[2], Michael Fink[2], Georg Gottlob[2], Luigi Granata[1], Gianluigi Greco[1], Edyta Kałka[4], Giovambattista Ianni[1], Domenico Lembo[3], Maurizio Lenzerini[3], Vincenzino Lio[1], Bartosz Nowicki[4], Riccardo Rosati[3], Marco Ruzzi[3], Witold Staniszkis[4], and Giorgio Terracina[1]

[1] Dipartimento di Matematica, Università della Calabria, Rende, Italy
[2] Inst. für Informationssysteme, Technische Universität Wien, Vienna, Austria
[3] Dip. di Informatica e Sistemistica, Università "La Sapienza", Roma, Italy
[4] Rodan Systems S.A., Warsaw, Poland

Abstract. The paper presents INFOMIX a successful application of ASP technology to the domain of Data Integration. INFOMIX is a novel system which supports powerful information integration, utilizing the ASP system DLV. While INFOMIX is based on solid theoretical foundations, it is a user-friendly system, endowed with graphical user interfaces for the average database user and administrator, respectively. The main features of the INFOMIX system are: (i) a comprehensive information model, through which the knowledge about the integration domain can be declaratively specified, (ii) capability of dealing with data that may result incomplete and/or inconsistent with respect to global constraints, (iii) advanced information integration algorithms, which reduce (in a sound and complete way) query answering to cautious reasoning on disjunctive Datalog programs, (iv) sophisticated optimization techniques guaranteeing the effectiveness of query evaluation in INFOMIX, (v) a rich data acquisition and transformation framework for accessing heterogeneous data in many formats including relational, XML, and HTML data.

1 Data Integration Systems

The enormous amount of information even more and more dispersed over many data sources, often stored in different heterogeneous formats, has boosted in recent years the interest for data integration systems (see, e.g, [2,6,5,4]). Roughly speaking, a data integration system $\mathcal{I}$ provides transparent access to different data sources by suitably combining their data, and providing the user with a unified view of them, called *global schema*. Formally, $\mathcal{I}$ can be seen as a triple $\langle \mathcal{G}, \mathcal{S}, \mathcal{M} \rangle$, where $\mathcal{G}$ is the global schema, $\mathcal{S}$ is the source schema, constituted by the schemas of all sources, and $\mathcal{M}$ is the mapping between $\mathcal{G}$ and $\mathcal{S}$ that contains a view over $\mathcal{S}$ for each element of $\mathcal{G}$. The users formulate their queries over $\mathcal{G}$, and the system automatically provides the answers. We assume that $\mathcal{G}$ and $\mathcal{S}$ are relational schemas, and that $\mathcal{G}$ contains integrity constraints (ICs) of three kinds: *key dependencies* (KDs), *exclusion dependencies* (EDs), and *inclusion dependencies* (IDs). Views in the mapping are specified in Datalog.

Example 1. Consider the integration system $\mathcal{I} = \langle \mathcal{G}, \mathcal{S}, \mathcal{M} \rangle$, where (i) $\mathcal{G}$ comprises the relation predicates $HasTutor(student, tutor)$ and $Student(name)$, (ii) $\mathcal{S}$ comprises

C. Baral et al. (Eds.): LPNMR 2005, LNAI 3662, pp. 379–383, 2005.
© Springer-Verlag Berlin Heidelberg 2005

the unary relation predicate s_1 and the binary predicates s_2 and s_3, (iii) $\mathcal{M}$ is defined by the Datalog program:

$$HasTutor(x, y) \leftarrow s_2(x, y) \vee s_3(x, y) \qquad Student(x) \leftarrow s_1(x).$$

ICs in $\mathcal{G}$ state that: (a) the key of $HasTutor$ is the attribute $student$ (KD); (b) a tutor cannot be a student (ED), and (c) each name in $Student$ must occur in $HasTutor$, i.e., each student has at least one tutor (ID). □

Data stored at the sources may violate global ICs when filtered through the mapping, since in general source data are not under the control of the data integration system. The standard approach to this problem is inherently procedural, and basically consists of explicitly modifying the data in order to eliminate violation of ICs (data cleaning). However, the explicit repair of data is not always convenient or possible. Therefore, when answering a user query, the system should be able to "virtually repair" relevant data in a declarative fashion (in the line of [1,2]) .

In this paper, we present INFOMIX, a novel system which supports powerful integration of inconsistent data by resorting to computational logic systems. The INFOMIX approach is based on the idea of reformulating a user query, taking into account the ICs of the global schema, in terms of a logic program Π, using disjunction, in a way such that the stable models of Π represent the repairs of the global database constructed by evaluating the mapping over the source extension. Then, answering a user query amounts to cautious reasoning over the logic program Π augmented by the facts retrieved from the sources. We point out that this reduction is possible since query answering in data integration systems is coNP-complete in data complexity in the above setting (and also in many other data integration frameworks [5,4]).

INFOMIX implements recent results of the database theory, which have been extended and specialized within the INFOMIX project [3,4,9,7,8,10]. While INFOMIX is based on solid theoretical foundations, it is a user-friendly system, endowed with graphical user interfaces for the average database user and administrator. INFOMIX is built in cooperation with RODAN systems, a commercial DBMS developer. Among the different features supported by the INFOMIX system,[1] we shall consider here the following two aspects:

- **Advanced information integration algorithms** [4] that reduce query answering to cautious reasoning on (head cycle free) disjunctive Datalog programs. This allows for effectively computing the query results (even in the presence of incomplete and/or inconsistent data) by using the state-of-the-art disjunctive Datalog system DLV [11]. The formal semantics of queries is captured also in the presence of incomplete and/or inconsistent data.
- **Sophisticated optimization techniques** [9,7,8] guarantee the effectiveness of query evaluation in INFOMIX. The novel optimization techniques, developed in INFOMIX, "localize" the computation and limit the inefficient (coNP) computation to a very small fragment of the input, obtaining fast query answering, even in such a powerful data integration framework.

[1] Further information, system documentation, and publications are available on the INFOMIX website: `www.mat.unical.it/infomix`.

We have experimented the system on a real-life application scenario. The results we have conducted clearly indicate the impact of the optimizations techniques on the system performance, and, more generally, show that, despite its worst-case computational complexity, the semantic approach to data integration pursued by INFOMIX can be effectively realized.

2 Logic Programming Within the INFOMIX Architecture

The INFOMIX system supports two modes: a design and a query mode. In the first, the global schema, the source schema, and the mapping between them are specified. Furthermore, wrappers for the data sources are created or imported. In the query mode, the system provides query answering facilities at run time, including data acquisition, integration, answer computation, and presentation to the user. In both the design and query mode, INFOMIX is conceptually divided into three levels, namely the *Information Service Level*, the *Internal Integration Level* and the *Data Acquisition and Transformation Level*. The most important level is the Internal Integration Level, which is based on computational logic and deductive database technology. It is composed by three modules, namely the Query Rewriter, the Query Optimizer and the Query Evaluator.

The *Query Rewriter* reformulates the user query taking into account global ICs. According to the notion of repair adopted in INFOMIX, the Query Rewriter can separately consider IDs and KDs and EDs in the reformulation process, as showed in [3,4]. More precisely, it makes use of a sub-module to verify data consistency w.r.t. KDs and EDs: exploiting the mapping, the sub-module unfolds the user query over the source relations and activates the corresponding wrappers to retrieve relevant data; then it checks wether there are KD or ED violations. If no violations occur, the reformulation produced by the rewriter is a simple (disjunction free) Datalog program, constructed according to the global IDs; otherwise, a suitable disjunctive Datalog program is generated that performs automatic repair of data w.r.t. both EDs and KDs. Basically, the Datalog program encodes the user query, the views in the mapping, and the global ICs that involve relation predicates that are relevant to answer the user query, in a way such that cautious answers to this program evaluated over the data sources correspond to the answers to the user query.

Example 2. In our example, consider now the the user query $q(x) \leftarrow HasTutor(x, y)$. Wrapper activation populates the Internal Data Store of the system with a database $\mathcal{D}$ for the source schema $\mathcal{S}$. Let us assume that the consistency check finds out that the KD and the EDs in $\mathcal{G}$ are violated by data migrating from the sources to the global schema through the mapping. Then, the Query Rewriter module reformulates the query in the following program:

$$\begin{aligned}
q(x) &:- \ HasTutor(x, y)\\
q(x) &:- \ Student(x)\\
HasTutor(x, y) &:- \ HasTutor_{\mathcal{D}}(x, y)\ ,\ not\ \overline{HasTutor}(x, y)\\
\overline{HasTutor}(x, y) \vee \overline{HasTutor}(x, z) &:- \ HasTutor_{\mathcal{D}}(x, y)\ ,\ HasTutor_{\mathcal{D}}(x, z)\ ,\ z \neq y\\
\overline{HasTutor}(x, y) \vee \overline{HasTutor}(y, z) &:- \ HasTutor_{\mathcal{D}}(x, y)\ ,\ HasTutor_{\mathcal{D}}(y, z)\\
Student(x) &:- \ Student_{\mathcal{D}}(x)\ ,\ not\ \overline{Student}(x)\\
\overline{HasTutor}(x, y) \vee \overline{Student}(y) &:- \ HasTutor_{\mathcal{D}}(x, y)\ ,\ Student_{\mathcal{D}}(y)\\
Student_{\mathcal{D}}(x) &:- \ s_1(x)
\end{aligned}$$

$$HasTutor_{\mathcal{D}}(x, y) :- s_2(x, y)$$
$$HasTutor_{\mathcal{D}}(x, y) :- s_3(x, y)$$

Informally, for each global relation r, the above program contains *(i)* a relation $r_{\mathcal{D}}$ that represents the extension of r obtained by evaluating the associated view in the mapping over the source database $\mathcal{D}$; *(ii)* a relation r that represents a subset of such extension that is consistent with the KD and the EDs for r; *(iii)* an auxiliary relation $\overline{r}$. The first two rules encode in the user query the ID stating that each student must have a tutor (intuitively, in order to return all the students that have a tutor, the rewriting looks also in the predicate $Student$). The third and the fourth rule encode the KD on $HasTutor$. The fifth, the sixth, and the seventh rule encode the ED stating that students cannot be tutors, whereas the last three rules encode the mapping $\mathcal{M}$. □

The *Query Optimizer* provides several optimization strategies, which turned out to be crucial for the efficiency of the system; in particular, the module exploits some *focusing* techniques which are able to isolate the portion of the source database that is relevant to answer the user query, by pushing constants in the query towards the sources. To this aim, an optimized (possibly disjunctive) Datalog program is generated by applying advanced binding propagation techniques à la Magic-Set [9,7].

Finally, the optimized program is passed to the *Query Evaluator*. It first loads data from the Internal Data Store and then invokes DLV [11] in order to compute the answers. The results are then sent to the *Information Model Manager* for suitable presentation to the user.

3 Application and Experiments

We have tested the INFOMIX prototype system on a real-life application scenario, in which data from various legacy databases and web sources must be integrated for a university information system. In particular, we built our information integration system on top of the data sources available at the University of Rome "La Sapienza".

The data sources comprise information on students, professors, curricula and exams in various faculties of the university. Currently, this data is dispersed over several databases in various (autonomous) administration offices and many webpages at different servers. Given this setting, we have devised a global schema of 14 relations and 29 integrity constraints, comprising KDs, IDs, and EDs.

The application scenario includes 3 legacy databases in relational format, comprising about 25 relations in total. The relation sizes range from a few hundred to tens of thousands of tuples (e.g., exam data). Besides these legacy databases, there are numerous web pages, which either provide information explicitly or through simple query interfaces (e.g., members of a department, phone numbers etc). We have developed a number of wrappers using LiXto tools [10], which extract information from these web sources. In total, there are about 35 data sources in the application scenario, which are mapped to the global relations through about 20 UCQs. Each UCQ joins up to three different logical data sources. Finally, we have formulated 9 typical queries with peculiar characteristics, which model different use cases.

Figure 1 shows the execution time for the 9 typical queries Q1, ...,Q9. Results are obtained on Pentium III machines running GNU/Linux with 256MB of memory. These

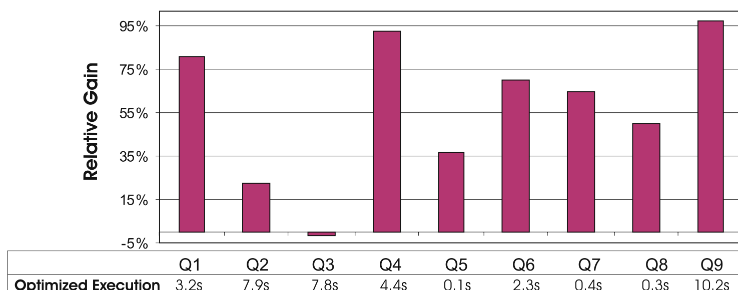


Fig. 1. Impact of Optimizations

demonstrate the feasibility of our approach, and the impact of optimization techniques. Actually, the figure shows the total execution time for the optimized INFOMIX system, where in particular a magic-set technique is included, and the relative gain w.r.t. the evaluation without any optimization. For many queries, we note a significant speed-up, especially for query Q9. We have verified that for Q9, the magic-set technique effectively prunes the unrelated conflicts, thus avoiding their repair. Note that for Q3 none of our optimizations applies; the overhead of the optimization methods (1%) is very lightweight, however.

Acknowledgment. This work was partially funded by the Information Society Technologies programme of the European Commission, Future and Emerging Technologies under the IST-2001-33570 INFOMIX project.

References

1. M. Arenas, L. E. Bertossi, and J. Chomicki. Consistent query answers in inconsistent databases. In *Proc. PODS 1999*, pages 68–79, 1999.
2. L. Bravo and L. Bertossi. Logic programming for consistently querying data integration systems. In *Proc. IJCAI 2003*, pages 10–15, 2003.
3. A. Calì, D. Lembo, and R. Rosati. On the decidability and complexity of query answering over inconsistent and incomplete databases. In *Proc. PODS 2003*, pages 260–271, 2003.
4. A. Calì, D. Lembo, and R. Rosati. Query rewriting and answering under constraints in data integration systems. In *Proc. IJCAI 2003*, pages 16–21, 2003.
5. J. Chomicki and J. Marcinkowski. Minimal-Change Integrity Maintenance Using Tuple Deletions. *Information and Computation*, 2004. 197(1-2): 90-121 (2005).
6. J. Chomicki, J. Marcinkowski, and S. Staworko. Computing consistent query answers using conflict hypergraphs. In *Proc. of CIKM 2004*, pages 417–426, 2004.
7. C. Cumbo, W. Faber, G. Greco, and N. Leone. Enhancing the magic-set method for disjunctive datalog programs. In *Proc. ICLP 2004*, pages 371–385, 2004.
8. T. Eiter, M. Fink, G. Greco, and D. Lembo. Efficient evaluation of logic programs for querying data int egration systems. In *Proc. ICLP 2003*, pages 163–177, 2003.
9. W. Faber, G. Greco, and N. Leone. Magic sets and their application to data integration. In *Proc. ICDT 2005*, 2005. LNCS 3363, in press.
10. G. Gottlob, C. Koch, R. Baumgartner, M. Herzog, and S. Flesca. The LiXto data extraction project - back and forth between theory and practice. In *Proc. PODS 2004*, pages 1–12, 2004.
11. N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The DLV System for Knowledge Representation and Reasoning. *ACM Trans. Comput. Log.*, 2004. To appear. Available via http://www.arxiv.org/ps/cs.AI/0211004.

Abduction and Preferences in Linguistics
Extended Abstract

Kathrin Konczak and Ralf Vogel

Institut für Informatik, Institut für Linguistik, Universität Potsdam,
Postfach 90 03 27, D–14439 Potsdam,
konczak@cs.uni-potsdam.de, rvogel@ling.uni-potsdam.de

Abstract. We associate optimality theory with abduction and preference handling. We present linguistic problems that appear in the study of dialects as new application of abduction and preference handling. Differences in German dialects originate from different rankings of linguistic constraints which determine the well-formedness of expressions within a language. We introduce a framework for analyzing differences in German dialects by abduction of preferences. More precisely, we will take the perspective of a linguist and reconstruct dialectal variation as abduction problem: Given an observation that a sentence is found as grammatically correct, abduct the underlying constraint ranking. For this, we give a new definition for the determination of optimal candidates for total orders with indifferences. Additionally, we give an encoding for the diagnosis front-end of the DLV system.

1 Background

We assume a basic familiarity with logic programs, answer set programming (ASP) [5], abduction (within ASP), and diagnosis [2,7,8,4].

In this work we want to find preference structures in a linguistic framework as explanations of an abduction problem. A linguistic grammar is a model of the implicit knowledge that guides linguistic behavior. This knowledge is usually conceived as a system of rules and/or well-formedness constraints which determine for a given language which expressions are well-formed and which are not. Our example is the verb order in 3-verb clusters of German dialects. A standard German clause with such a 3-verb cluster looks as follows:

Maria glaubt, dass sie das Lied <u>*singen*</u> <u>*müssen*</u> <u>*wird*</u>.
Maria thinks that she the song sing must will.

Swiss and Standard German follow different ordering strategies for the verbs:

	Default verb cluster orders
Standard German:	*singen müssen wird*
Swiss German:	*wird müssen singen*

Syntactic structures are composed recursively by *complementation*. The object, here: "*ein Lied*" is the complement of its governing head, here: the predicative verb "*singen*". This verb phrase, "*ein Lied singen*", is the complement of the modal verb "*müssen*", and this modal verb phrase, in Standard German: "*ein*

C. Baral et al. (Eds.): LPNMR 2005, LNAI 3662, pp. 384–388, 2005.
© Springer-Verlag Berlin Heidelberg 2005

Lied singen müssen", is the complement of the temporal auxiliary verb, "*werden*". The differences between the languages and variants can now be described in terms of complement-head order:

	Default complement-head orders
Standard German:	All complements precede their heads: "*ein Lied singen müssen wird*"
Swiss German:	Noun complements precede their heads, verbal complements follow them: "*wird müssen ein Lied singen*"
English	All complements follow their heads: "*will have to sing a song*"

The differences between the three languages can be reconstructed within Optimality Theory (OT) using the following three constraints:

H-Comp A complement follows its head.
Comp-H A complement precedes its head.
H-VComp A verbal complement follows its head.

Constraint rankings are indicated with "$\gg$", meaning "has higher priority than". The three rankings that conform to the observations are the following: [1]

Standard German: *Comp-H* $\gg$ *H-Comp* (*H-VComp*)
Swiss German: *H-VComp* $\gg$ *Comp-H* $\gg$ *H-Comp*
English: *H-Comp* $\gg$ *Comp-H* (*H-VComp*)

The exact rank of *H-VComp* can only be determined in Swiss German. While its effects are completely subsumed by the high rank of *H-Comp* in English, in Standard German all that is necessary is that *Comp-H* has highest priority, while the relative order of *H-Comp* and *H-VComp* is irrelevant because of the low rank of these two constraints. Grammars are usually, but not necessarily, strict total orders of constraints. The rankings given here are only the crucial ones. For those which are left open, any order will be compatible with the observations.

2 Optimal Candidates

A linguistic grammar predicts the well-formedness of expressions. Optimality theory grammars do so by establishing a competition between different candidate expressions which are evaluated on a hierarchy of well-formedness constraints. An OT grammar is an input-output mapping. The input defines what is to be expressed. We then have a set of candidate output expressions. The candidates incur different constraint violations. Each candidate is evaluated on the basis of the constraint hierarchy, and the candidate that performs best in this evaluation is the winner, the optimal, hence, grammatical expression.

In the following, we will consider the determination of optimal candidates wrt well-formedness of expressions. For this, let $\mathcal{X}$ be a set of candidates (sentences), $\mathcal{C}$ be a set of constraints, $\delta : \mathcal{X} \times \mathcal{C} \rightarrow I\!N$ be a violation function, where $\delta(x, c)$ denotes the degree of violation of $x \in \mathcal{X}$ wrt $c \in \mathcal{C}$, and $\ll$ be a (strict) total order on $\mathcal{C}$. Then, we call $\mathcal{L} = (\mathcal{X}, \mathcal{C}, \delta, \ll)$ a linguistic framework.

[1] These rankings were found out by empirical linguistic studies.

In Section 1, we have taken an example for the dialectal variation in German 3-verb clusters. In the following we will elaborate this example. For the 3-verb cluster we have the following possible word orders:[2] Maria glaubt, dass sie das Lied ...

(321) singen müssen wird. **(231)** müssen singen wird.
(123) wird müssen singen. **(132)** wird singen müssen.
(312) singen wird müssen. **(213)** müssen wird singen.

Our set of candidates $\mathcal{X} = \{321, 231, 123, 132, 312, 213\}$ is constituted by these possible word orders. [3] $\mathcal{C} = \{H\text{-}VComp, Comp\text{-}H, H\text{-}Comp\}$ is our set of constraints. The degree of violation denotes how well a sentence fulfills a constraint. It is represented by the number of asterisks ($*$), as in the following table.

	H-VComp	*Comp-H*	*H-Comp*
321	$***$		$**$
231	$**$	$*$	$*$
123		$**$	
132	$*$	$*$	$*$
312	$**$	$*$	$**$
213	$*$	$**$	$*$

Next, we want to clarify when a sentence is a best candidate wrt a given constraint ranking [1].

Definition 1. *Let* $\mathcal{L} = (\mathcal{X}, \mathcal{C}, \delta, \ll)$ *be a linguistic framework, where* $\ll$ *is a strict total order on* $\mathcal{C}$*. Then, candidate* $x \in \mathcal{X}$ *is a* winner *if there does not exist a candidate* $y \in \mathcal{X}, x \neq y$ *such that there exists a constraint* $c \in \mathcal{C}$ *where*
1. for all $c' \in \mathcal{C}$ *where* $c \ll c'$ *we have that* $\delta(x, c') \geq \delta(y, c')$*, and*
2. $\delta(x, c) > \delta(y, c)$*.*

In our example, for the constraint order $H\text{-}VComp \gg Comp\text{-}H \gg H\text{-}Comp$ (Swiss German), we get 123 as winner. For constraint ranking $Comp\text{-}H \gg H\text{-}VComp \gg H\text{-}Comp$ (Standard German), candidate 321 is a winner.

3 Abduction of Constraint Rankings

In Optimality theoretic terms, linguists observe that a candidate is determined by a speaker as a *winner*, expressing that the sentence is grammatically correct. The problem is that the observer does not know which underlying constraint ranking the speaker has. Here, abduction comes into play. Given a linguistic framework $\mathcal{L} = (\mathcal{X}, \mathcal{C}, \delta)$ with an unknown constraint ranking $\ll$ and an observation that candidate $x \in \mathcal{X}$ wins, we want to abduct $\ll$ which explains x.

The set of candidates is given by rules (1) $cd(x) \leftarrow$ for each $x \in \mathcal{X}$, the set of constraints by (2) $cst(c) \leftarrow$ for each $c \in \mathcal{C}$, and the violation degrees by (3) $viol(x, c, \delta(x, c)) \leftarrow$ where $\delta(x, c)$ is the degree of violation of $x \in \mathcal{X}$ wrt $c \in \mathcal{C}$.

[2] The numbers signal the hierarchical position of the verb. Verb 1 is the temporal auxiliary (*werden*), verb 2 the modal (*müssen*), and verb 3 the predicative (*singen*).
[3] Due to OT we have to consider all possibilities.

According to Definition 1, a winner, can be determined by the following rules:

(4) $winner(X) \leftarrow cd(X), not\ defeated(X)$
(5) $defeated(X) \leftarrow cd(X), cd(Y), Y \neq X, better(Y, X)$
(6) $better(Y, X) \leftarrow cd(X), cd(Y), Y \neq X, cst(C), wins(Y, X, C), not\ hp(X, Y, C)$
(7) $hp(X, Y, C) \leftarrow cd(X), cd(Y), Y \neq X, cst(C), pref(C_1, C), wins(X, Y, C_1)$
(8) $wins(X, Y, C) \leftarrow cd(X), cd(Y), cst(C), viol(X, C, NX), viol(Y, C, NY), NX < NY$
(9) $pref(X, Z) \leftarrow pref(X, Y), pref(Y, Z)$
(10) $\leftarrow pref(C, C), cst(C)$
(11) $\leftarrow cst(C_1), cst(C_2), unranked(C_1, C_2), C_1 \neq C_2$
(12) $unranked(C_1, C_2) \leftarrow not\ pref(C_1, C_2), not\ pref(C_2, C_1), cst(C_1), cst(C_2)$

Our logic program Π consists of the rules (1)–(12). [4] An observation is that exactly one candidate is observed as a winner, but the other candidates not [1]:

$$O(x) = \left\{ \begin{array}{l} winner(x) \leftarrow \text{for } x \in \mathcal{X} \\ defeat(y) \leftarrow \ \text{for all } y \in \mathcal{X}, y \neq x \end{array} \right\}$$

Our hypothesis is the set of all possible pairwise preferences: $H = \{pref(c, c') \leftarrow \mid c, c' \in \mathcal{C}, c \neq c'\}$.

Then, an explanation $\Delta \subseteq H$ for $\langle \Pi, H, O \rangle$ gives us a possible strict total order among the constraints such that $\Pi \cup \Delta$ explains O. More precisely, $\Delta \subseteq H$ is an explanation if $O(x) \subseteq S$ for some answer set S of $\Pi \cup \Delta$.

In our linguistic example we have additionally the background knowledge that constraint *Comp-H* is strictly higher preferred than *H-Comp*. Hence, we have $pref(comph, hcomp) \leftarrow$. additionally in Π. Then, our hypotheses is $H = \{pref(comph, hvcomp) \leftarrow, pref(hvcomp, hcomp) \leftarrow, pref(hvcomp, comph) \leftarrow, pref(hcomp, hvcomp) \leftarrow\}$, which gives us together with $pref(comph, hcomp) \leftarrow$ all possible constraint rankings. For computing explanations, we use `DLV` [3,4] with the command `-FD` for abductive diagnosis. As a result, we get that 321 has two explanations, $\Delta_1 = \{Comp\text{-}H \gg H\text{-}Comp \gg H\text{-}VComp\}$ and $\Delta_2 = \{Comp\text{-}H \gg H\text{-}VComp \gg H\text{-}Comp\}$. 123 has one explanation, $\Delta = \{H\text{-}VComp \gg Comp\text{-}H \gg H\text{-}Comp\}$, and for observing other candidates as winners, we get no explanation. This means that the constraints are not sufficient for explaining these candidates as a winner. Candidate 321 yields two explanations, $H\text{-}Comp \gg H\text{-}VComp$ and $H\text{-}VComp \gg H\text{-}Comp$. This supposes that *H-Comp* and *H-VComp* can be ranked equally. Since Def. 1 is only valid for (strict) total orders, we have to extend it for total (pre-)orders.

Definition 2. *Let $\mathcal{L} = (\mathcal{X}, \mathcal{C}, \delta, \preceq)$ be a linguistic framework, where $\preceq$ is a total order on $\mathcal{C}$. Then, candidate $x \in \mathcal{X}$ is a* winner *if there exists no $y \neq x$ such that there exists a $c \in \mathcal{C}$ such that*

1. for all $c' \neq c$ such that $c' \approx c$ or $c' \succ c$ we have $\delta(c', x) \geq \delta(c', y)$, and
2. $\delta(c, y) < \delta(c, x)$.

The encoding for Definition 1 is adapted to Definition 2 in a straight forward way. Our hypotheses are now the set of all pairwise strict preference relations and the set of all possible indifferences: $H = \{pref(c, c') \leftarrow \mid c, c' \in \mathcal{C}, c \neq c'\} \cup \{prefeq(c, c') \leftarrow \mid c, c' \in \mathcal{C}, c \neq c'\}$. From this set of hypotheses, all possible

[4] Note that the abducted preference ordering must be total, as required in OT.

total orders are constructible. Coming back to our example, for candidate 321, we additionally get the supposed explanation that $H\text{-}VComp \approx H\text{-}Comp$.

4 Discussion and Further Work

We have associated OT with abduction and preference handling. Abduction and preference handling were studied in many other issues, e.g. in [6].But, as far as the authors know, abduction and preferences were not yet linked to OT before.

We have shown that abduction within ASP is a useful knowledge reasoning tool for linguistic problems, here: dialectic studies, where the abducted explanations match the empirical results found out by the linguists. We have taken the perspective of a linguist and have reconstructed dialectal variation as abduction problem: Given an observation that a sentence is found as grammatically correct, abduct the underlying constraint ranking of the dialect. Furthermore, we have provided an encoding (within ASP) for the diagnosis front-end of `DLV`.

Regarding linguistic studies, there is an ongoing debate about how unique the rule systems of language are in human cognition, as well as in biology in a very broad sense. The reconstruction of grammatical regularities with abduction and preference handling has consequences for this debate: if grammars can be modeled this way, then they share core properties with other non-linguistic rule systems. This supports a position that does not make special assumptions about the nature of linguistic rule systems.

Further work is to study results when observing non-unique optimal candidates and to study abduction of partial orders instead of total orders, since partial orders may be enough for explaining the observations.

Acknowledgments. The first author was supported by the German Science Foundation (DFG) under grant SCHA 550/6, TP C and by the EC under project IST-2001-37004 WASP. The second author was supported by the DFG under grant FOR-375/2-A3.

References

1. P. Besnard, G. Fanselow, and T. Schaub. Optimality theory as a family of cumulative logics. *Journal of Logic, Language and Information*, 12(2):153–182, April 2003.
2. M. Denecker and A. Kakas. Abduction in logic programming. In A. Kakas and F. Sadri, editors, *Computational Logic. Logic Programming and Beyond*, volume 2407 of *Lecture Notes in Computer Science*, pages 402–436. Springer-Verlag, 2002.
3. `DLV`. `http://www.dbai.tuwien.ac.at/proj/dlv/`.
4. T. Eiter, W. Faber, N. Leone, and G. Pfeifer. The diagnosis frontend of the DLV system. *AI Communications*, 12(1-2):99–111, 1999.
5. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. pages 1070–1080. The MIT Press, 1988.
6. K. Inoue and C. Sakama. Abducing priorities to derive intended conclusions. *IJCAI-99*, pages 44–49. Morgan Kaufmann Publishers Inc., 1999.
7. A. Kakas, R. Kowalski, and F. Toni. The role of abduction in logic programming. In *Handbook of logic in Artificial Intelligence and Logic Programming*, volume 5, pages 235–324. Oxford University Press, 1998.
8. A. C. Kakas and P. Mancarella. Generalized stable models: A semantics for abduction. *ECAI'90*, pages 385–391, 1990.

Inference of Gene Relations from Microarray Data by Abduction

Irene Papatheodorou, Antonis Kakas*, and Marek Sergot

Department of Computing, Imperial College London, SW7 2AZ, UK
{ivp, ack, mjs}@doc.ic.ac.uk

Abstract. We describe an application of Abductive Logic Programming (ALP) to the analysis of an important class of DNA microarray experiments. We develop an ALP theory that provides a simple and general model of how gene interactions can cause changes in observable expression levels of genes. Input to the procedure are the observed microarray results; output are hypotheses about possible gene interactions that explain the observed effects. We apply and evaluate our approach on microarray experiments on *M. tuberculosis* and *S. cerevisiae.*

1 Introduction

The focus in bioinformatics has shifted from the analysis of genome sequences, now available in their entirety for several organisms, to functional genomics, which seeks to ascribe biological function to genes and understand gene interactions. An important tool in these studies is DNA microarray technology, which enables simultaneous measurement of expression levels of thousands of genes. A common form of experiment measures differences in expression levels of whole genomes in differing environmental conditions and/or after deletion or overexpression of one or more genes. The aim is to obtain clues about gene interactions and unravel pathways that define the cell's responses to various stimuli. The datasets generated are too large and complex for manual analysis. Raw data are analysed using statistical techniques to define significantly differentially expressed genes. Methods for further interpretation of the results, in terms of gene interactions, remain largely undeveloped, however, though Bayesian Networks have recently attracted attention (See e.g. [1] for an overview).

We formulate the analysis of this type of microarray data as a problem of *abduction*, that is, inference from observable effects, i.e. the microarray data, to possible causes, hypotheses about possible gene interactions. We construct an Abductive Logic Program (ALP) theory which provides a simple, general model of how gene interactions can cause changes in observable expression levels of genes—essentially a formalisation of the (usually implicit) reasoning used by biologists designing microarray experiments. Adjustable parameters allow us to constrain the search for hypotheses and apply the methods to large data sets. A novel feature of our method is the ability to deal with observations in many separate experiments together.

* Visiting from Department of Computer Science, University of Cyprus.

C. Baral et al. (Eds.): LPNMR 2005, LNAI 3662, pp. 389–393, 2005.
© Springer-Verlag Berlin Heidelberg 2005

The model is validated by comparing the inferred hypotheses against known gene interactions and by assessing the biological plausibility of the hypotheses where detailed information is lacking. We use microarray data sets on *M. tuberculosis* and *S. cerevisiae* (yeast). Section 3 presents an example of inferences that re-discover part of the *M. tuberculosis* heat shock response pathway.

There are many issues and other experimental methods in the search for gene regulation mechanisms. To our knowledge, inference of gene networks from microarray data has not previously been formulated as a problem of abduction, though abduction has been used in [6] to construct a genetic network from classical genetics experiments. The nature of the data, the hypotheses and the model itself, differ from what is addressed here.

2 The Model

Input to the procedure is a set of observations expressed as logic assertions of the form *increases_expression(Expt, Gene)* and *reduces_expression(Expt, Gene).* They are obtained by statistical analysis of the raw microarray data to determine the significance of measured differences of expression levels of each gene [4].

The output is a set of abducible relations of two different types: *induces (Gene1, Gene2)* and *inhibits(Gene1,Gene2)* for the hypothesis that *Gene1* induces the expression of *Gene2*, or inhibits it, respectively. Each individual experiment provides partial clues about possible *induces/inhibits* relations between genes.

The modelling framework we employ is Abductive Logic Programming (ALP) [2], an extension of logic programming that allows declarative logical representations of the problem domain and supports abductive reasoning. A theory is represented by a triple (P, A, IC), where P is a logic program, A a set of abducible predicates and IC a set of classical logic formulae, the *integrity constraints*.

An abductive explanation for a query Q is a set Δ of ground abducible atoms on the predicates A such that: $P \cup \Delta \models_{LP} Q$, $P \cup \Delta$ is consistent, $P \cup \Delta \models_{LP} IC$, where $\models_{LP}$ denotes a standard entailment relation of logic programming.

The integrity constraints IC impose additional *validity* requirements on the hypotheses Δ. They are modularly stated in the theory, in addition to the basic model in P. They augment any partial information on the abducible predicates or impose other constraints on the abductively generated explanations. We form integrity constraints (IC) of three different types: *(1) self-consistency:* For example, a gene cannot both inhibit and induce the same gene at the same time (under the same conditions). *(2) consistency with background information:* Background knowledge, such as known inhibitor genes can also be expressed in the form of *IC*. *(3) experimental consistency:* When analysing the results of an experiment E in which a gene G is mutated, we may want to consider as 'intermediary genes' (see below) only genes whose expression is also observed to be affected in E.

Gene Interactions

Top-level Rules The program P of the ALP theory represents how gene interactions can increase or reduce the expression of genes, as observed in the experiments. An assumption is that such observed variations in gene expression

should be attributed directly or indirectly to the variations (gene mutations or environmental stress), carried out in the experiment(s) investigated. This assumption is not justified for all classes of microarray experiments or all forms of gene regulation.

For example: if an experiment E knocks out a gene G, and G inhibits gene X, then E will show an increased expression of X — subject to some possible exceptions. This rule is expressed in logic programming notation as follows:

increases_expression(E, X) ← (1)
knocks_out(E, G), inhibits(G, X),
not incr_affected_by_other_gene(E,G,X),
not incr_affected_by_EnvFact(E, X).

E is a variable that ranges over names of experiments and G, X are variables that represent genes. *increases_expression(E,X)* is observational data from the experiment E, *inhibits(G,X)* is part of the unknown information to be abduced, and *knocks_out(E,G)* provides background knowledge about the experiment E.

The last two conditions express possible exceptions that deal with the possibility that the difference in gene expression can be attributed to a factor other than the mutated gene: e.g. (a) a gene other than G, or (b) an environmental factor. Here, *not* is the logic programming construct 'negation as failure', used to express that (1) is a default general rule subject to the stated exceptions.

Similarly, there is a rule that deals with the cases of reduced expression of G in experiment E. Similar rules cover the cases of over-expressing G and further rules deal with the various combinations of gene mutation and changes in environmental conditions according to our classification of experiment types.

Rule (1) only accounts for direct relationships between the mutated gene and the differentially expressed one. These relationships could be indirect: Inference of intermediate steps of interaction is achieved by further recursive rules:

increases_expression(E, X) ← (2)
mutates(E, G), intermediary_gene(E, Gx, G),
reduces_expression(E, Gx), inhibits(Gx, X),
not incr_affected_by_other_gene(E,Gx,X),
not incr_affected_by_EnvFact(E, X).

If gene Gx inhibits gene X, and the expression of gene Gx is reduced (directly or indirectly) by the mutation of gene G in experiment E, then the expression of X is increased in the experiment E. The relation *mutates(E,G)* covers both knock-out and over-expression of gene G in the experiment E.

The Parameters are relations that control the genes taken into account when searching for hypotheses. In the general case, where every gene is possibly related to other genes, there may be an exponential number of possible hypotheses. With the parameters we constrain the problem by reducing the search space. By varying their definition, we can test different possibilities of the model. There are two parametric relations including *intermediary_gene/3* in rule (2). The integrity constraints provide another means of constraining the search space.

3 Application: Heat Shock Response of *M. tuberculosis*

M. tuberculosis data sets were obtained from our collaborators at the Centre for Microbiology and Infection, Imperial College London [4] and publicly available tables from the Schoolnik lab, Stanford University. Observations and inferred hypotheses are presented as directed graphs using a set of visualisation tools we developed, which is based on *Graphviz*, an open source graph-layout software from the AT&T Laboratories. A web-based front-end to explore and manipulate the graphical displays is available [3].

In the example shown here, observations from 5 experiments were selected according to a conservatively chosen significance threshold in the 1st phase statistical analysis. Each experiment knocks out or over-expresses a gene believed to be involved in heat shock response and known to function as a transcriptional regulator (regulator of expression of other genes). The two parameters of the model were defined to restrict attention to possible interactions between 16 genes of known regulatory function. Analysis of the observations in all experiments together generated a single hypothesis, shown in graphical form in figure [1], that explained all observations.

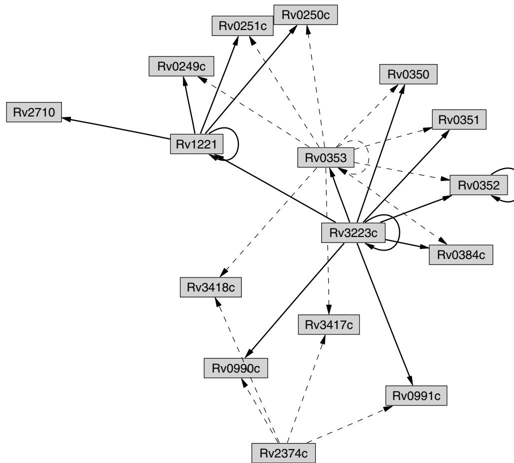


Fig. 1. The nodes represent genes, whereas the edges show the inferred relations between genes, bold for *induces* and dotted for *inhibits*. Cyclical edges represent the auto-regulation relationships abduced.

The resulting hypothesis is in agreement with previous knowledge [4]. The DnaK operon[1] (genes Rv0350–353, on the right of the figure) is controlled by the positive regulator sigH (Rv3223c) and the negative regulator hspR (Rv0353). The acr2 operon (genes Rv0249c–251c, at the top left of the figure) is controlled by the positive regulator sigE (Rv1221) and negative regulator hspR (Rv0353).

[1] Group of genes that reside next to each other on the DNA and are expressed together.

The groES/EL genes (Rv3417c–Rv3418c) are under dual negative control by hspR (Rv0353) and hrcA (Rv2374c). Known feedback loops are also discovered: the DnaK operon (Rv0350–353) is negatively regulated via its member Rv0353. Finally, there is a group of genes whose function in heat shock response is not clear but are linked in the explanatory hypothesis. Rv0249c and Rv0250c are both unknown genes, repressed (inhibited) by hspR, both next to Rv0251c in the chromosome. This could be a real effect, suggesting they are in an operon, or it could be some artefact due to their place on the chromosome and the way data is collected. Similarly, Rv0990c and Rv0991c could also be members of an operon, but isolated with no obvious function in heat shock. Our collaborators are planning to investigate these hypotheses in a new set of experiments. Further discussion of our methods and more detailed examples are available in [5].

4 Conclusions

We develop a *general method* to support the analysis of an important class of microarray experiments. The novel feature is a simple, general model of how gene interactions can cause changes in observable expression levels of genes under differing conditions, and the use of abduction to infer explanatory hypotheses. This method allows us to infer regulation relations across several experiments.

The declarative and modular nature of this *gene interaction model* allows us to experiment easily with variations and new general rules suggested by our biological collaborators, and to add biological knowledge as it becomes available. The parameters in the model allow us to constrain the search space of possible hypotheses and thereby apply the methods to realistically large data sets.

Tests on *M. tuberculosis* rediscovered part of the heat shock response mechanism and suggested further experiments. We are presently engaged in a systematic exploration of the various possibilities afforded by the model and an extensive validation against known gene regulation processes in yeast.

We have been able to apply these methods in practice to the analysis of large data sets. Whatever the biological significance of this technique turns out to be in the long-term, the model provides a valuable test case for those concerned with the development of abductive reasoning technology.

References

1. Friedman N., Linial M., Nachman I., Pe'er (2000) Using Bayesian Networks to analyze expression data. *J. Comp. Bio.* 7:601–620.
2. Kakas, A.C., Denecker, M. (2002) Abduction in Logic Programming. In *Computational Logic: Logic Programming and Beyond, Part I.* Springer-Verlag, pp402–436.
3. Papatheodorou I., Sergot M., Randall M., Stewart G., Robertson B. (2004) Visualisation of Microarray results to assist interpretation *Tuberculosis* 84:275–281
4. Stewart, G. *et al* (2002) Dissection of the heat-shock response in *Mycobacterium Tuberculosis* using mutants and microarrays. *Microbiology* 10:3129–3138.
5. Technical Report No. 2005/3, Department of Computing, Imperial College London, 2005
6. Zupan, B., Demsar, J., Bratko, I. *et al* (2003) GenePath: a system for automated construction of genetic networks from mutant data. *Bioinformatics* 19:383–389.

nomore$^<$: A System for Computing Preferred Answer Sets

Susanne Grell, Kathrin Konczak, and Torsten Schaub

Institut für Informatik, Universität Potsdam, Postfach 90 03 27, D–14439 Potsdam
sgrell@rz.uni-potsdam.de, {konczak, torsten}@cs.uni-potsdam.de

1 Introduction

The integration of preferences into Answer Set Programming (ASP) constitutes an important practical device for distinguishing certain preferred answer sets from non-preferred ones. Up to now, the preference semantics we are considering in this system description were incorporated into answer set solvers either by meta-interpretation [3] or by pre-compilation front-ends [2]; therefore, such kinds of preferences were never integrated into the core existing ASP solvers.

Unlike this, the `nomore`$^<$ system pursues an integrative approach to preference handling. Its theoretical background is described in [6]. The system itself is an early branch of the `nomore++` ASP solver [8]. The approach relies on rule dependency graphs and computes answer sets by coloring this graph by following a certain strategy. It integrates preferences as an additional type of edges among nodes representing rules. The idea is to start from an uncolored graph and to employ specific operators that turn a partially colored graph gradually into a totally colored one that represents a preferred answer set.

Apart from describing the `nomore`$^<$ system, we focus in our experimental section on the question how an integrative approach compares to a compilation-based approach. To this end, we use the `plp` system [9] in connection with `nomore`$^<$ as well as `smodels` [11] as its respective back-end ASP solvers. (Ab)using `nomore`$^<$ as a standard ASP solver allows us to obtain comparable results, although its performance is much more inferior than that of `smodels` as well as the full-fledged `nomore++` system [8].

2 Background

The current version of the `nomore`$^<$ system computes preferred answer sets of logic programs with preferences on rules. A *logic program* is a finite set of rules such as $p_0 \leftarrow p_1, \dots, p_m, not\ p_{m+1}, \dots, not\ p_n$, where $n \geq m \geq 0$, and each p_i $(0 \leq i \leq n)$ is an *atom*. The semantics of logic programs is determined by its set of answer sets, as proposed in [5]. An *ordered program* is a pair $(\Pi, <)$, where Π is a logic program and $< \subseteq \Pi \times \Pi$ is a strict partial order among the rules in Π. Given, $r_1, r_2 \in \Pi$, the relation $r_1 < r_2$ expresses that r_2 has *higher priority* than r_1. The ordering $<$ is used for selecting preferred answer sets among the standard ones of Π, which can be achieved in different ways. Here, we concentrate on one

C. Baral et al. (Eds.): LPNMR 2005, LNAI 3662, pp. 394–398, 2005.
© Springer-Verlag Berlin Heidelberg 2005

interpretation for preference handling, namely the D-semantics provided in [2]. Similar semantics for preferences among rules where defined in [1,10].

Previous work [6] provides the theoretical background for computing preferred answer sets in terms of coloring strategies of a rule dependency graph extended by preference information. The main goal of the work presented in [6] is the integration of preference information into an ASP solver. There, deterministic operators, e.g. $\mathcal{U}, \mathcal{P}$ and nondeterministic operators, e.g. choice operator $\mathcal{D}$, where defined. For instance, iterated application of $\mathcal{P}$ (denoted by $\mathcal{P}^*$) and $\mathcal{U}$, denoted by $(\mathcal{PU})^*$, yield the well-founded semantics [12]. Iterated application of propagation $(\mathcal{PU})^*$ and choice operations $\mathcal{D}$, denoted by $[(\mathcal{PU})^* \circ \mathcal{D}]^n \circ (\mathcal{PU})^*$, yield preferred answer sets of the underlying ordered program. More precisely, these propagation and choice operations are preference-based, following the idea that rules are considered in an order preserving way.

3 System

The `nomore<` system is a C++ implementation of the approach given in [6] and an improvement of the `GCplp` [4] system. The current version 1.0 can be downloaded from `http://www.cs.uni-potsdam.de/wv/nomorepref/`. Since `nomore<` uses `lparse` [7] as parser, we have to encode preference statements as the following set of rules:

$$\begin{array}{rl} name(r) \leftarrow & \text{for all rules } r \text{ involved in } < \\ preferred(r_1, r_2) \leftarrow & \text{for every preference relation } r_2 < r_1, \end{array}$$

where the name predicates *name* are needed for rule labelling and the *preferred* predicates make the preferences explicit. For example, the ordered program $\{r_1 : a \leftarrow not\ b,\ r_2 : b \leftarrow not\ a,\ r_2 < r_1\}$ is represented as program

$$\begin{array}{ll} a \leftarrow name(r_1), not\ b & name(r_1) \leftarrow \\ b \leftarrow name(r_2), not\ a & name(r_2) \leftarrow \\ & preferred(r_1, r_2) \leftarrow \end{array}$$

which has standard answer sets $\{a\}$ and $\{b\}$, but only $\{a\}$ as preferred one.

4 Experiments

We have considered the following examples:

- $indset_\langle N \rangle.lp$ encodes the independent set problem for an undirected circle graph with N vertices, where even numbered vertices are preferred to be in an independent set.
- $art2_\langle N \rangle.lp$ and $art_\langle N \rangle.lp$ are artificial ordered programs, where all N rules of the underlying logic program have only negative body atoms (except for *name* predicates used for rule labelling).

- *kernel_comp_*$\langle N \rangle$*.lp* encodes the kernel problem for a complete graph with N vertices, where one special vertex is preferred to be in a kernel, but no other ones.
- *col_lad_1_*$\langle N \rangle$*.lp* encodes the coloring problem of a ladder graph with two colors, where a particular color is preferred for every odd vertex.

A detailed description of the used examples including downloads can be found at `http://www.cs.uni-potsdam.de/~konczak/benchmarks/BenchPref/index.html`.

Table 1 shows the time measurements on an AMD processor with 2.2 Ghz and 2 GB memory. We have tested several problem classes for finding one preferred answer set and finding all preferred answer sets. In Table 1, DH denotes a coloring strategy, where the propagation operators are preference-based (i.e. we propagate from higher preferred rules to lower preferred rule) and the choice operator is a conventional one. There, generated solutions have to be checked by the operator $\mathcal{H}$, which verifies the existence of a so called *height function* [6] ensuring that an answer set is a preferred one. $D^<$ denotes a coloring strategy, where the propagation operators and the choice operators are fully preference-based. More precisely, it denotes the coloring sequence $[(\mathcal{PU})^* \circ \mathcal{D}]^n \circ (\mathcal{PU})^*$ [6]. That is, propagating and choosing goes along the given preferences from higher preferred rules to lower preferred ones. In short, DH offers a partial integration of preference information into an ASP solver where a check is still needed, whereas $D^<$ fully integrates preference information into an ASP solver.

In contrast to the integration of preferences into an ASP solver, the `plp` system [9] compiles an ordered program into a logic program such that the preferred answer sets correspond to the standard answer set of the compiled program. We have used `plp` to compare our integrative approach with the compilation method provided by `plp`. In Table 1, $plp + n$ denotes the time for computing preferred answer sets via the `plp` compilation while using the `nomore`$^<$ system as a standard ASP solver. Additionally, we have run the compilation in connection with `smodels` [11] (see $plp + s$ in Table 1) for showing differences in the current development status of the `nomore`$^<$ system, when computing standard answer sets.

The times for $plp + s$ and $plp + n$ show that the `nomore`$^<$ system has a lower base speed than `smodels` due to the fact that `nomore`$^<$ is not as optimized as smodels, e.g. heuristics and lookahead are not yet integrated in `nomore`$^<$ and `nomore`$^<$ provides only forward propagation whereas `smodels` (as well as `nomore++` [8]) provides forward and backward propagation.

The results in Table 1 show that for problems where one is interested in finding only one solution, the strategy where preference information is fully integrated into an ASP solver (indicated by appealing to the preference-based choice operator $D^<$) is much better than a strategy where the preference information is only partially integrated. Additionally in this case, $D^<$ behaves much better than the compilation method ($plp + n$). $D^<$ is also a good strategy for determining preferred answer sets whenever the underlying rules of the program have no positive body atoms, e.g. as in the artificial examples *art2_*$\langle N \rangle$*.lp*. The strategy DH seems to be good whenever we want to find all preferred answer sets or whenever we have multiple positive atoms in the body of an rule.

Table 1. Time Measurements on an AMD processor with 2.2 Ghz and 2 GB memory, and `lparse` version 1.0.13 and `smodels` version 2.28

	find one preferred answer set				find all preferred answer sets			
file	DH	D$^<$	plp+n	plp+s	DH	D$^<$	plp+n	plp+s
indset_5.lp	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
indset_10.lp	0.00	0.00	0.02	0.01	0.00	0.00	0.02	0.01
indset_15.lp	0.01	0.00	0.06	0.01	0.02	0.02	0.06	0.01
indset_20.lp	0.00	0.00	0.13	0.02	0.14	0.18	0.13	0.02
indset_25.lp	0.29	0.01	0.23	0.04	0.68	0.89	0.23	0.04
indset_30.lp	0.01	0.01	0.38	0.05	3.42	8.47	0.38	0.05
indset_35.lp	6.83	0.01	0.59	0.07	16.25	41.90	0.60	0.07
indset_40.lp	0.01	0.01	0.88	0.09	78.31	371.60	0.90	0.09
art2_10.lp	0.00	0.00	0.02	0.01	0.00	0.00	0.05	0.01
art2_16.lp	0.02	0.00	0.05	0.02	0.03	0.00	0.32	0.02
art2_20.lp	0.06	0.00	0.09	0.02	0.10	0.00	0.82	0.02
art2_30.lp	1.44	0.01	0.27	0.05	2.55	8.05	4.91	0.05
art2_40.lp	32.81	0.01	0.62	0.08	58.40	0.01	19.08	0.08
art2_50.lp	–	0.02	1.36	0.13	–	–	69.11	0.13
art2_60.lp	–	0.02	2.57	0.19	–	–	–	0.19
art2_70.lp	–	0.03	4.26	0.25	–	–	–	0.25
art2_80.lp	–	0.04	6.50	0.33	–	–	–	0.33
art2_90.lp	–	0.05	9.32	0.42	–	–	–	0.43
art2_100.lp	–	0.06	12.74	0.52	–	–	–	0.52
kernel_comp_10.lp	0.03	0.01	0.11	0.03	0.03	0.34	0.11	0.03
kernel_comp_20.lp	0.21	0.06	0.70	0.11	0.21	1087.77	0.72	0.11
kernel_comp_30.lp	0.81	0.16	2.87	0.25	0.80	–	2.93	0.25
kernel_comp_40.lp	2.20	0.33	7.09	0.46	2.17	–	7.23	0.46
kernel_comp_50.lp	5.03	0.60	14.37	0.73	4.99	–	14.23	0.73
kernel_comp_60.lp	10.34	1.01	24.28	1.07	9.91	–	24.88	1.07
kernel_comp_70.lp	21.02	1.60	38.54	1.46	20.00	–	39.00	1.46
kernel_comp_80.lp	46.69	2.55	59.49	1.92	41.67	–	59.11	1.91
kernel_comp_90.lp	87.17	3.87	84.36	2.45	85.27	–	83.31	2.44
col_lad_1_2.lp	0.00	0.00	0.00	0.00	0.00	0.00	0.01	0.00
col_lad_1_4.lp	0.00	0.00	0.06	0.01	0.00	0.05	0.06	0.01
col_lad_1_8.lp	0.01	0.01	3.13	0.02	0.01	33.78	3.16	0.02
col_lad_1_10.lp	0.01	0.01	19.38	0.03	0.02	–	19.47	0.03
col_lad_1_20.lp	0.05	0.06	–	0.10	0.07	–	–	0.10
col_lad_1_30.lp	0.11	0.13	–	0.22	0.16	–	–	0.22
col_lad_1_40.lp	0.20	0.23	–	0.38	0.28	–	–	0.39
art_10.lp	0.02	0.00	0.06	0.01	0.03	0.00	0.07	0.01
art_20.lp	37.24	0.04	0.70	0.04	55.93	0.04	0.84	0.04
art_30.lp	–	0.29	3.63	0.10	–	0.29	4.41	0.10
art_40.lp	–	1.21	15.18	0.18	–	1.19	18.94	0.18
art_50.lp	–	3.60	39.98	0.28	–	3.74	47.92	0.28
art_60.lp	–	9.01	–	0.41	–	8.90	–	0.41
art_70.lp	–	19.24	–	0.57	–	19.89	–	0.57
art_80.lp	–	38.40	–	0.77	–	37.78	–	0.77
art_90.lp	–	70.38	–	1.00	–	68.47	–	0.99
art_100.lp	–	121.81	–	1.26	–	116.74	–	1.25

For the independent set problem, the compilation method ($plp + n$) seems to be better than an integration of preference information into an ASP solver. This is mainly caused by the underlying preference structure. It remains to be future work, under which preference structure the compilation method and under which preference structure the integrative approach yields the best results. Hence, choosing the best strategy for computing preferred answer set depends on the underlying problem (example) class.

5 Conclusions and Future Work

We have presented the `nomore`$^<$ system, implemented in C++, which integrates preference information into an ASP solver. Furthermore, we have compared our integrative approach with the compilation method of preference handling provided by the `plp` system. We have found out that it really depends on the underlying problem class and preference structure, whether an integrative approach of preference handling is better than the compilation method, or vice versa. Hence, it remains to be future work to study problem classes and underlying preference structures in view of compilation and integrative methods. Moreover, the current version of `nomore`$^<$ contains no optimization methods, e.g. backward propagation, heuristics, and lookahead.

Acknowledgements. The work was supported by the German Science Foundation (DFG) under grant SCHA 550/6-4, TP C and by the EC under project IST-2001-37004 WASP.

References

1. G. Brewka and T. Eiter. Preferred answer sets for extended logic programs. *Artificial Intelligence*, 109(1-2):297–356, 1999.
2. J. Delgrande, T. Schaub, and H. Tompits. A framework for compiling preferences in logic programs. *Theory and Practice of Logic Programming*, 3(2):129–187, 2003.
3. T. Eiter, W. Faber, N. Leone, and G. Pfeifer. Computing preferred answer sets by meta-interpretation in answer set programming. *Theory and Practice of Logic Programming*, 3(4-5):463–498, 2003.
4. `GCplp`. `http://www.cs.uni-potsdam.de/~konczak/system/GCplp`.
5. M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991.
6. K. Konczak, T. Schaub, and T. Linke. Graphs and colorings for answer set programming with preferences. *Fundamenta Informaticae*, 57(2-4):393–421, 2003.
7. `Lparse`. `http://saturn.tcs.hut.fi/Software/smodels/`.
8. `nomore++`. `http://www.cs.uni-potsdam.de/wv/nomore++/`.
9. `plp`. `http://www.cs.uni-potsdam.de/~torsten/plp`.
10. T. Schaub and K. Wang. A semantic framework for preference handling in answer set programming. *Theory and Practice of Logic Programming*, 3(4-5):569–607, 2003.
11. `smodels`. `http://www.tcs.hut.fi/Software/smodels/`.
12. A. van Gelder, K. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.

Integrating an Answer Set Solver into Prolog: ASP – PROLOG

Omar Elkhatib, Enrico Pontelli, and Tran Cao Son

Department of Computer Science,
New Mexico State University
{okhatib, epontell, tson}@cs.nmsu.edu

1 Introduction

A number of answer set solvers have been proposed in recent years, such as *Smodels*, DLV, Cmodels, and ASSAT. Most existing ASP solvers have been extended to provide front-ends that are suitable to encode specialized forms of knowledge—e.g., weight-constraints, restricted forms of optimization, front-ends for planning and diagnosis. These features allow declarative solutions in specific application domains. However, this is not completely satisfactory:

- The development of an ASP program is viewed as a "monolithic" process. Most ASP systems offer only a batch approach to execution of programs—programs are completely developed, "compiled", executed, and finally answer sets are proposed to the user. The process lacks any levels of interaction with the user. In particular, it does not directly support an interactive development of programs (as in Prolog), where one can immediately explore the results of simply adding/removing rules.
- ASP programmers can control the computation of answer sets through the rules that they include in the logic program. Nevertheless, ASP systems offer very limited capabilities for reasoning on the *whole class* of answer sets associated to a program—e.g., to perform selection of models according to user-defined criteria or to compare models. These activities are important in many application domains—e.g., to express soft constraints, to support preferences when using ASP to perform planning.
- ASP solvers are independent systems; interaction with other languages can be performed only through complex, low level APIs; this prevents programmers from writing programs that manipulate ASP programs and answer sets as first-class citizens. E.g., we wish to write programs in a high-level language (Prolog in this case), which are capable to access ASP programs, modify their structure (by adding or removing rules), and access and reason with answer sets. This type of features is essential in many domains—e.g., automatically modify the plan length in a planning problem.

We address these problems by developing a system, ASP – PROLOG. The system provides a tight integration of ASP in Prolog. The language is developed using the module and class capabilities of CIAO Prolog. ASP – PROLOG allows

C. Baral et al. (Eds.): LPNMR 2005, LNAI 3662, pp. 399–404, 2005.
© Springer-Verlag Berlin Heidelberg 2005

programmers to assemble a variety of different modules to create a program; along with the traditional types of modules supported by CIAO Prolog (e.g., standard Prolog, constraint logic programming, active deductive databases), it allows the presence of various *ASP modules*, each being a logic program conforming to the syntax of **lparse**. Each Prolog module can access any ASP modules (using the traditional *module qualification* of Prolog), read its content, access its models, and modify it (e.g., adding/removing rules).

2 System Capabilities and Possible Areas of Application

User Interface: Prolog modules are required to declare their intention to access any ASP modules; this is accomplished through the declarations

:− `use_asp(`*`module_name`*`, `*`file_name`*`, `*`parameters`*`)`

where *module_name* is the name of the ASP module, *file_name* is the file containing the ASP code, while *parameters* control the ASP solver (command line and `compute` arguments of SMODELS). ASP − PROLOG provides predicates that allow Prolog to interact with ASP modules:

- `model(ModelName,ModelObject)` retrieves one answer set of an ASP module; `ModelName` is an atom uniquely identifying one answer set, while `ModelObject` is a CIAO Prolog object containing the answer set (as Prolog facts). For example, `plan:model(1, Q)` retrieves the answer set named `1` of ASP module `plan` and stores it as an object in `Q`; if we want to check whether the atom `p` is true in such answer set, we simply execute the Prolog goal `Q:p`.
- *`total_stable_model/2`* determines the number of answer sets of an ASP module, and returns a list of the names given to the answer sets.
- `assert/1` and `retract/1`: the argument of these predicates is a list of ASP rules, that are either added or removed from an ASP module. For example, the goal `plan:assert([p:-q])` adds the rule `p:-q` to the ASP module `plan`. Modifications are undone during backtracking.
- `assert_nb/1` and `retract_nb/1` have the same effect as `assert/retract`, with the exception that the modifications are not undone upon backtracking.
- `change_parm/1` allows to set/modify the parameters for the ASP execution (e.g., values of constants, components of the `compute` statement of SMODELS).
- `clause/2`: this predicate is used to allow a Prolog module to access the rules of an ASP module—in the same spirit as the `clause` predicate is employed in Prolog to access the Prolog rules present in the program. The two arguments represent respectively the head and the body of the rule.

If α is a CIAO object representing an answer set, then the Prolog goal $\alpha : p$ corresponds to testing truth of p (possibly non-ground) in the answer set α. Observe that, due to the fact that the syntax of *Smodels* is not ISO Prolog-compliant, certain *Smodels* constructs (e.g., cardinality and weight constraints) have a slightly different syntactic representation when used within Prolog modules.

Possible Application Areas: ASP − PROLOG allows users to

- *manipulate answer sets of a program*; this includes *(i)* the computation of the entailment relation of a logic program (e.g., different modes of reasoning, such as *skeptical* or *credulous* reasoning can be done), and *(ii)* comparing answer sets of an ASP program, and select those that satisfy certain properties (e.g., individual preferences).
  Since ASP has frequently been used as a knowledge representation language, the ability to compute the entailment relation of a logic program makes ASP − PROLOG an attractive candidate for the implementation of query-answering systems based on answer set semantics. Furthermore, because computing preferred answer sets has found its application in planning with preferences, diagnosis, and common-sense reasoning, ASP − PROLOG can be used as an interactive environment for such systems.
- *modify programs and recompute their answer sets on the fly* (e.g., adding or removing rules); this feature provides a simple way to modify the values of constants occurring in a program and/or to delay the grounding process of a program with variables until the instantiated rules are needed. Furthermore, ASP − PROLOG provides different ways for users to test and debug an ASP program interactively. For example, 'suspected rules' can be removed for testing the consistency of a program; adding 'known-to-be-true' atoms into a program is another way for detecting errors in the program; etc.
- *work with several logic programs simultaneously* (e.g., reasoning about the common knowledge of multiple-agents); since ASP − PROLOG allows users to work with several modules at the same time, it can be used as an environment for implementing/testing various formalisms for modeling multi-agents. For example, when each agent's knowledge is represented by a logic program, computing common knowledge among them is equivalent to computing the intersection of all possible answer sets.
- use logic programs with answer set semantics to control the query-answering process of a Prolog program (e.g., avoiding infinite loops); this is possible, since ASP − PROLOG programs are Prolog programs extended with a new type of atoms, whose truth value is determined by the ASP solver.

We are not aware of any systems with the same capabilities as ASP − PROLOG. *Smodels* provides a very low level API [5] that allows C++ programs to use *Smodels* as a library. DLV does not document any external API, although a Java wrapper has been recently announced [1]. XASP [2] proposes an interface from XSB to the API of *Smodels*. It provides a subset of the functionalities of ASP − PROLOG, with a deeper integration with the capabilities of XSB of handling normal logic programs.

3 System Implementation

The syntax and semantics of ASP − PROLOG programs are described in [3]. It suffices to notice that a ASP − PROLOG program is a pair (Pr, As), where Pr is a set of ASP − PROLOG rules and As is a logic program which conforms to

the syntax of **lparse**. Each $\mathbb{ASP}-\mathbb{PROLOG}$ rule is a Prolog rule whose body can contain atoms occurring in As. Furthermore, the above mentioned interface predicates (e.g., `assert, retract`) can be used by $\mathbb{ASP}-\mathbb{PROLOG}$ programs to manipulate As. The overall structure of the implementation is depicted in Figure 1. The system is composed of two parts, a *preprocessor* and the actual CIAO Prolog system. The system accepts the input composed of *(i)* the main Prolog module (Pr); *(ii)* a collection of CIAO Prolog modules $(m_1, m_2, \ldots, m_n)$; *(iii)* a collection of ASP modules $(e_1, e_2, \ldots, e_m)$.

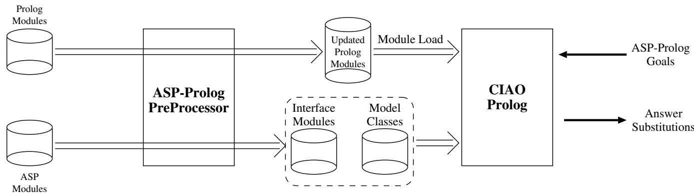


Fig. 1. Overall Structure of $\mathbb{ASP}-\mathbb{PROLOG}$ Implementation

Preprocessing: The input to the system is used as the input to the preprocessor. The preprocessor transforms each Prolog module into a new module (Pr is transformed to NPr and m_i is transformed to nm_i), and each ASP module e_i into a CIAO module im_i and a class definition c_i.[1] The main purpose of this step is to adapt the syntax of the interface predicates, make it compatible with CIAO Prolog's syntax, and prepare different *objects* (interface module and model class) for the actual program execution. The preprocessor also invokes the CIAO Prolog top-level and loads all the appropriate modules for execution. The interaction with the user is the same as the standard Prolog top-level.

Interface Modules: For each ASP module e_i, the preprocessor generates an interface module c_i by instantiating a generic module skeleton to the content of e_i. c_i is a standard CIAO Prolog module and provides the client Prolog modules with the predicates used to access and manage the ASP module e_i. The overall structure of the interface module is illustrated in Figure 2. The module has an export list which includes all the predicates used to manipulate ASP modules (e.g., `assert`, `retract`, `model`) as well as all the predicates that are defined within the ASP module. The typical module declaration generated for an interface module will look like:

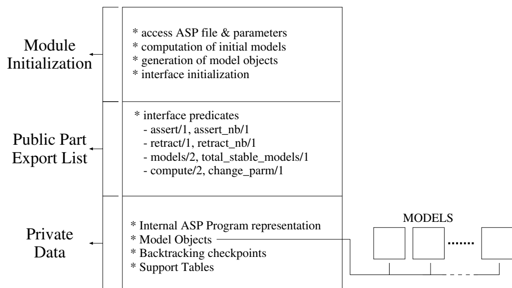


Fig. 2. Structure of the Interface Module

[1] CIAO provides the ability to define classes and create class instances [4].

:– `module('plan.lp',[assert/1,retract/1,...,model/2,p/0,q/0,r/0]).` The definition of the various exported predicates (except for the predicates defined in the ASP module) is derived by instantiating a generic definition of each predicate. Each module has an initialization part, which is in charge of setting up the internal data structures (e.g., the internal representation of the ASP module, tables to store parameters and answer sets), and invoke the answer set solvers for the first time on the ASP module—in the current prototype we are using *Smodels* as answer set solver. The result of the computation of the models will be encoded as a collection of *Model Objects* (see below). The module will maintain a number of internal data structures, including a representation of the ASP code, a representation of the parameters to be used for the computation of the answer sets (e.g., values of constants), a list of the objects representing the models of the ASP module, and a count of the current number of answer sets.

Model Classes: For each ASP module e_i, the preprocessor generates a CIAO class definition c_i. The objects obtained from the instantiation of such class will be used to represent the individual models of the ASP module. Prolog modules can obtain reference to these objects (e.g., using the `model` predicate supplied by the interface module) and use them to directly query the content of one or several models. The definition of the class is obtained through a straightforward parsing of the ASP module, to collect the names of the predicates defined in it; the class will provide a public method for each of the predicates present in the ASP module. In addition, the class defines also a public method `add/1` which is employed by the interface module to initialize the content of the model.

Each model is stored in one instance of the class; the actual atoms representing the model are stored internally in the objects as facts of the form $s(\langle \texttt{fact} \rangle)$.

Implementation and System Details: The various interface predicates are implemented in CIAO Prolog in a fairly straightforward way. For instance, the implementation of `assert` (resp. `retract`) makes use of the `module_concat` of CIAO Prolog to introduce new (resp. remove) rules to (resp. from) the ASP module.

A number of tables are maintained by each interface module to support the execution of ASP modules. Some of the relevant internal structures include:

- *fn:* maintains a (Prolog-based) representation of the rules of the ASP module;
- *stable_ref:* a table (implemented as Prolog facts) that maintains references to the current answer sets of the ASP module (as pairs *model name*/*object reference* that maps name of models to objects representing the models);
- *retract_rule:* a trail structure that caches the modifications performed by `assert` and `retract`; this is required to allow undoing of the changes;
- *prm:* a table (encoded as Prolog facts) that stores the parameters to be used during the computation of the models of the ASP module.

Code and URL: The original idea and the semantics of $\mathbb{ASP} - \mathbb{PROLOG}$ has been presented in [3]. The first version of system is now complete and available for download at `www.cs.nmsu.edu/~okhatib/asp_prolog.html`.

References

1. The DLV Wrapper Project. `160.97.47.246:8080/wrapper`, 2003.
2. L. Castro. XASP: Answer Set Programming with XSB. SUNY Stony Brook, 2002.
3. O. Elkhatib et al. ASP-Prolog: A System for Reasoning about Answer Set Programs in Prolog. In *PADL-2004*, pages 148–162. Springer, 2004.
4. M. Pineda et al. The O'Ciao Approach to Object Oriented Logic Programming. In *Colloquium on Implementation of Constraint Logic Programming Systems*, 2002.
5. T. Syrjänen. Lparse User's Manual. `www.tcs.hut.fi/Software/smodels`.

CIRC2DLP — Translating Circumscription into Disjunctive Logic Programming*

Emilia Oikarinen** and Tomi Janhunen

Helsinki University of Technology,
Department of Computer Science and Engineering,
Laboratory for Theoretical Computer Science,
P.O.Box 5400, FI-02015 TKK, Finland
{Emilia.Oikarinen, Tomi.Janhunen}@tkk.fi

1 Introduction

The *stable model semantics* of disjunctive logic programs (DLPs) is based on minimal models [5,12] which makes atoms appearing in a disjunctive program false by default. This is often desirable from the knowledge representation point of view, but certain domains become awkward to formalize if all atoms are blindly subject to minimization. In contrast to this, *parallel circumscription* [11] provides a refined notion of minimal models as it distinguishes *varying* and *fixed* atoms in addition to those being falsified. This eases the task of knowledge presentation in many cases. For example, it is straightforward to formalize Reiter-style *minimal diagnoses* [13] for digital circuits using parallel circumscription.

There have been several attempts to embed parallel circumscription into disjunctive logic programming. Although fixed atoms are easy in this respect [1,6], varying atoms are not fully covered. Earlier approaches either deal with syntactic subclasses of logic programs [4] or have exponential worst-case space complexities [9,14]. To the contrary, the system CIRC2DLP described in this article is based on a new *linear* but *non-modular* transformation [8] which enables the use of existing implementations of disjunctive logic programming such as DLV [10] and GNT [7] for the actual search of minimal models.

The rest of this system description is organized as follows. Section 2 concentrates on specifying the output produced by CIRC2DLP, i.e. the translation presented in [8], on a high level of abstraction. In Section 3, we provide the reader with some instructions how to use the tool in practice. The last section (Section 3) comprises of some preliminary experimental results obtained using CIRC2DLP together with DLV and GNT. As a benchmark, we use the problem of finding Reiter-style minimal diagnoses [13] for digital circuits. Moreover, we briefly compare the performance of our translation-based approach with another: CIRCUM [15] is a system developed for computing *prioritized circumscription* (a generalization of parallel circumscription) using a *generate* and *test* method.

* The research reported in this paper is partially funded by the Academy of Finland (project #211025) and the European Commission (contract IST-FET-2001-37004).
** The support from Helsinki Graduate School in Computer Science and Engineering, Nokia Foundation, and Finnish Cultural Foundation is gratefully acknowledged.

C. Baral et al. (Eds.): LPNMR 2005, LNAI 3662, pp. 405–409, 2005.
© Springer-Verlag Berlin Heidelberg 2005

2 Translation-Based Approach

The aim in the following is to briefly describe the translation computed by CIRC2DLP. For that purpose, we give a definition for parallel circumscription in the propositional case. Following the presentation in [8], we formulate the definition in the case of a *positive* DLP Π possessing a Herbrand base $\mathrm{Hb}(\Pi)$. Given a set of *varying* atoms $V \subseteq \mathrm{Hb}(\Pi)$ and a set of *fixed* atoms $F \subseteq \mathrm{Hb}(\Pi)$, the parallel circumscription of Π is characterized by $\langle V, F\rangle$-*minimal* models $M \models \Pi$ each of which is minimal in the following sense: there is no model $N \models \Pi$ such that $N \setminus (V \cup F) \subset M \setminus (V \cup F)$ and $N \cap F = M \cap F$.

The basic idea in the translation-based approach [8] is that the $\langle V, F\rangle$-minimal models of a positive DLP Π can be captured by projecting the stable models of the translation

$$\mathrm{Tr}_{\mathrm{circ2dlp}}(\Pi) = \mathrm{Tr}_{\mathrm{GEN}}(\mathrm{Tr}_{\mathrm{KK}}(\Pi)) \cup \mathrm{Tr}_{\mathrm{EG}}(\mathrm{Tr}_{\mathrm{MIN}}(\mathrm{Tr}_{\mathrm{KK}}(\Pi))) \quad (1)$$

with respect to $\mathrm{Hb}(\Pi)$. The translation given in (1) is based on the following primitives: (i) $\mathrm{Tr}_{\mathrm{KK}}(\cdot)$ removes fixed atoms using a technique proposed in [1], (ii) $\mathrm{Tr}_{\mathrm{GEN}}(\cdot)$ produces a model generator for Π as a DLP, (iii) $\mathrm{Tr}_{\mathrm{MIN}}(\cdot)$ encodes a test for $\langle V, \emptyset\rangle$-minimality as a propositional unsatisfiability problem, and (iv) $\mathrm{Tr}_{\mathrm{EG}}(\cdot)$ implements the required unsatisfiability check using the primitives of DLPs [3]. To summarize the properties of $\mathrm{Tr}_{\mathrm{circ2dlp}}(\Pi)$, the translation is linear in the length of the program $||\Pi||$ and a bijective correspondence between stable models of the translation and the $\langle V, F\rangle$-minimal models of Π is obtained. Further details and the correctness proof of $\mathrm{Tr}_{\mathrm{circ2dlp}}(\Pi)$ can be found in [8].

The model generator $\mathrm{Tr}_{\mathrm{GEN}}(\mathrm{Tr}_{\mathrm{KK}}(\Pi))$ can be enhanced by taking the set of varying atoms V properly into account. Actually, an improved model generator $\mathrm{Tr}_{\mathrm{GEN2}}(\mathrm{Tr}_{\mathrm{KK}}(\Pi))$ is already used in the implementation. The idea is to replace the rules in items 1 and 2 in [8, Definition 6] by the following:

1′. $a \leftarrow \sim\overline{a}$ for each $a \in V$ and $\overline{a} \leftarrow \sim a$ for each $a \in \mathrm{Hb}(\Pi)$,
2′. $(A \setminus V) \leftarrow (B \setminus V), \sim(A \cap V), \sim\overline{B \cap V}$ for each rule $A \leftarrow B$ in Π.

The translation $\mathrm{Tr}_{\mathrm{circ2dlp}}(\Pi)$ optimized in this way is a DLP and thus a valid input for disjunctive solvers implementing the search for stable models [7,10].

Prioritized circumscription [11] can be translated into parallel circumscription using a scheme proposed by Lifschitz [11]:

$$\mathrm{Circ}(\Pi, P_1 > \ldots > P_k, V) = \bigwedge_{i=1}^{k} \mathrm{Circ}(\Pi, P_i, P_{i+1} \cup \cdots \cup P_k \cup V) \quad (2)$$

where $P_1, \ldots, P_k$ are sets of minimized atoms with decreasing priority, V is the set of varying atoms, and the sets of fixed atoms remain implicit. In our translation-based approach, the equation (2) is understood as a join of k parallel circumscriptions. The respective subtranslations can be concatenated so that $\mathrm{Hb}(\Pi)$ is shared and the new atoms produced by $\mathrm{Tr}_{\mathrm{circ2dlp}}$ remain distinct for each part. It follows that the time complexity is $\mathcal{O}(k \times ||\Pi||)$ in general.

3 Some Instructions for Use

The translator CIRC2DLP has been implemented in C under Linux operating system and is available at `http://www.tcs.hut.fi/Software/circ2dlp/`. The translator takes a DLP Π combined with sets of varying and fixed atoms as input and produces translation $\mathrm{Tr}_{\mathrm{circ2dlp}}(\Pi)$ as output. The input format is the internal format of GNT produced by the front-end LPARSE. Rules with variables can be used, although LPARSE performs an instantiation for the rules. CIRC2DLP produces output compatible with both GNT (default) and DLV.

All atoms are minimized by default, unless explicitly stated to be varying or fixed. Default behaviour can be altered using option `--vary`. Notice that LPARSE might produce *invisible* atoms that have no name in the symbol table. Option `--vary` cannot be applied to programs containing invisible atoms, as the semantics of invisible atoms becomes unclear. CIRC2DLP can also handle programs containing negation. For such programs the translation yields the $\langle V, F\rangle$-minimal models of the Gelfond-Lifschitz reduct of the original program which can be understood as the $\langle V, F\rangle$-stable models of the program.

Command line options for CIRC2DLP are the following:

- `-h` or `--help` – Print a help message.
- `-t` – Print human readable output.
- `--dlv` – Print the output in DLV syntax.
- `--vary` – Vary all atoms by default.
- `--all` – Generate all classical model candidates, using the model generator $\mathrm{Tr}_{\mathrm{GEN}}(\mathrm{Tr}_{\mathrm{KK}}(\Pi))$. Otherwise, $\mathrm{Tr}_{\mathrm{GEN2}}(\mathrm{Tr}_{\mathrm{KK}}(\Pi))$ is used.
- `--version` – Print version information.

For example, all $\langle\{a\}, \emptyset\rangle$-minimal models of a program stored in a file `example.lp` can be computed as follows:

```
lparse --dlp example.lp > example.sm
circ2dlp example.sm -v a | gnt 0
```

or, with DLV,

```
circ2dlp --dlv example.sm -v a | dlv -n=0 --
```

For more examples, see `http://www.tcs.hut.fi/Software/circ2dlp/`.

The translator CIRC2DLP is accompanied by a Perl implementation of Lifschitz's scheme for computing prioritized circumscription called PRIO_CIRC2DLP. For example, the script is used to compute prioritized circumscription $\mathrm{Circ}(\Pi, \{a\} > \{b\}, \emptyset)$ for program Π given in file `example.sm` as follows:

```
prio_circ2dlp example.sm a:b | gnt 0
```

4 Experiments

As a benchmark, we use the problem of finding Reiter-style minimal diagnoses [13] for digital circuits encoded as parallel circumscription. We generate circuits

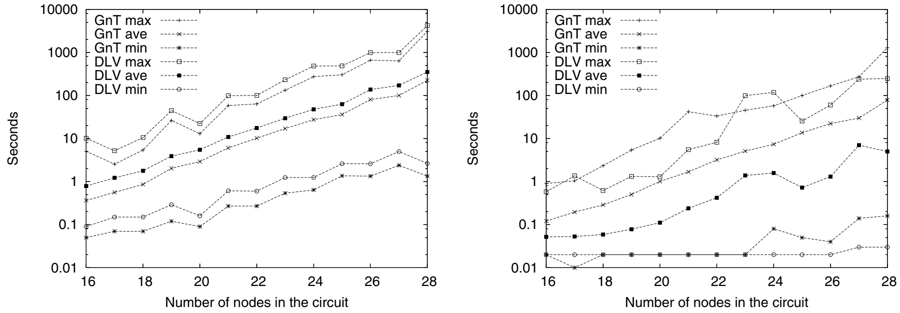


Fig. 1. Computing minimal diagnoses for faulty digital circuits. On the left *all* diagnoses are computed, whereas only *one* diagnosis on the right.

as follows. First, a random tree is generated using the inverse Pruefer algorithm. The leaves of the tree corresponding to the inputs of the digital circuit are assigned random Boolean values. The intermediate nodes are assigned random logical operations corresponding to the intermediate gates of the circuit. The gate at the root node of the tree produces the output of the circuit. The value of the output is calculated and flipped in order to obtain faulty behaviour for the circuit. The number of nodes N in the tree forming the digital circuit varies from 16 to 28. For each number of nodes we generate 100 random instances. These instances are available at `http://www.tcs.hut.fi/Software/circ2dlp/`. Typically an instance has less than ten minimal diagnoses when $N = 28$.

The measured running time is the translation time of CIRC2DLP plus the duration of the search for stable models using GNT or DLV. The actual translation times are negligible, however. We use user+system time of `/usr/bin/time` command in UNIX. All the tests were run under the Debian GNU/Linux 2.4.26 operating system on a AMD Athlon XP 2000+ computer with 1 GB memory.

Results are illustrated in Fig. 1. In the case of finding *all* minimal diagnoses, GNT outperforms DLV, but in the case of finding a *single* minimal diagnosis DLV is superior to GNT in most of the cases. One obvious advantage of our translation-based approach is that it is rather easy to use different solvers and thus gain from their development in the future.

We also compared briefly the performance of our approach with that of the CIRCUM system [15] using some instances of our diagnosis benchmark. Our experiments suggest that in the case of parallel circumscription the running times for the CIRCUM system are one or two orders of magnitude higher than running times for CIRC2DLP+GNT. To compare the systems in the case of prioritized circumscription we used our diagnosis benchmarks and added random priorities for the minimized atoms varying the number of priority classes k. These experiments suggest that our approach is able to compete with CIRCUM when k is small, but as k grows, the quadratic blowup implied by (2) becomes apparent.

There is also a diagnosis front-end in DLV [2], but there are restrictions in the case of minimal diagnoses: the theory has to be non-disjunctive and the

abnormality atoms may only appear negatively. This limits the applicability of the front-end so that our diagnosis benchmark cannot be represented naturally.

References

1. J. de Kleer and K. Konolige. Eliminating the fixed predicates from a circumscription. *Artificial Intelligence*, 39(3):391–398, July 1989.
2. T. Eiter, W. Faber, N. Leone, and G. Pfeifer. The diagnosis frontend of the DLV system. *AI Communications*, 12(1-2):99–111, 1999.
3. T. Eiter and G. Gottlob. On the computational cost of disjunctive logic programming: Propositional case. *Annals of Math. and AI*, 15:289–323, 1995.
4. M. Gelfond and V. Lifschitz. Compiling circumscriptive theories into logic programs. In *Proceedings of the 7th National Conference on Artificial Intelligence*, pages 455–449, St. Paul, MN, August 1988. AAAI Press.
5. M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991.
6. K. Inoue and C. Sakama. Negation as failure in the head. *Journal of Logic Programming*, 35(1):39–78, 1998.
7. T. Janhunen, I. Niemelä, D. Seipel, P. Simons, and J.-H. You. Unfolding partiality and disjunctions in stable model semantics. *ACM Transactions on Computational Logic*, 2005. To appear, see `http://www.acm.org/tocl/accepted.html`.
8. T. Janhunen and E. Oikarinen. Capturing parallel circumscription with disjunctive logic programs. In *Proceedings of the 9th European Conference on Logics in Artificial Intelligence, JELIA'04*, pages 134–146, Lisbon, Portugal, September 2004. Springer-Verlag. LNAI 3229.
9. J. Lee and F. Lin. Loop formulas for circumscription. In *Proceedings of 19th National Conference on Artificial Intelligence*, pages 281–286, San Jose, California, USA, July 2004. The MIT Press.
10. N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, and F. Scarcello. The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic*, 2005. To appear, see `http://www.acm.org/tocl/accepted.html`.
11. V. Lifschitz. Computing circumscription. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, pages 121–127, Los Angeles, California, USA, August 1985. Morgan Kaufmann.
12. T.C. Przymusinski. Stable semantics for disjunctive programs. *New Generation Computing*, 9:401–424, 1991.
13. R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32:57–95, 1987.
14. C. Sakama and K. Inoue. Embedding circumscriptive theories in general disjunctive programs. In *Proceedings of the 3rd International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 344–357. Springer-Verlag, 1995.
15. T. Wakaki and K. Inoue. Compiling prioritized circumscription into answer set programming. In *Proceedings of the 20th International Conference on Logic Programming*, pages 356–370, Saint-Malo, France, September 2004. Springer-Verlag.

Pbmodels — Software to Compute Stable Models by Pseudoboolean Solvers

Lengning Liu and Mirosław Truszczyński

Department of Computer Science, University of Kentucky,
Lexington, KY 40506-0046, USA

Abstract. We describe a new software, *pbmodels*, that uses pseudo-boolean constraint solvers (PB solvers) to compute stable models of logic programs with weight atoms. To this end, *pbmodels* converts ground logic programs to propositional theories with weight atoms so that stable models correspond to models. Our approach is similar to that used by *assat* and *cmodels*. However, unlike these two systems, *pbmodels* does not compile the weight atoms away. Preliminary experimental results on the performance of *pbmodels* are promising.

1 Introduction

We describe a system *pbmodels* to compute stable models of logic programs with weight atoms. We call such programs *smodels* programs, as we adopt for them the semantics implemented in *smodels* [1]. The key idea behind *pbmodels* is to translate programs into propositional theories and use propositional satisfiability solvers. However, unlike existing systems *assat* [2] and *cmodels* [3], which first exploited this direction, we do not replace weight atoms with propositional formulas. Instead, we translate programs with weight atoms directly into theories in propositional logic *extended* with weight atoms, which we refer to as the logic PL^{wa} [4]. We then use existing solvers for such theories as computational back-end engines. We refer to solvers testing satisfiability of PL^{wa} theories as PB solvers. In some cases, prior to the use of a PB solver, additional simple transformations are needed to ensure the right input format.

Compiling away weight atoms may lead to significantly larger programs and theories. Currently most advanced translations result in the growth by a logarithmic factor in the case of cardinality atoms, and a polynomial factor in the case of general weight atoms. Moreover, for some solvers, especially those based on the local-search idea, the structure of the resulting theories makes them difficult to process. The growth in size and the additional structure, which result from compiling weight atoms away, often render solvers that require that step, such as *assat* [2] and *cmodels* [3], less effective. That motivates our work. *Pbmodels* is designed to compute models of *smodels* programs *without* replacing weight atoms by their pure propositional representations.

2 *Pbmodels* — The Algorithm

Our paper is based on theoretical results developed in [4] for an abstract setting of programs with monotone and convex constraints. The approach developed there specializes

C. Baral et al. (Eds.): LPNMR 2005, LNAI 3662, pp. 410–415, 2005.
© Springer-Verlag Berlin Heidelberg 2005

to *smodels* programs. In particular, it yields the concepts of the program completion, a loop, a terminating loop and a loop formula for an *smodels* program, with the completion and loop formulas being formulas in the logic PL^{wa}.

Results in [4] imply that stable models of an *smodels* program are in one-to-one correspondence with models (in the sense of the logic PL^{wa}) of the completion of the program extended with some loop formulas for the program. That result, generalizing a result of Lin and Zhao [2], allows us to extend the design of *assat* to the case of *smodels* programs in a way, which does not require that weight atoms be complied away. In a nutshell, we first compute the completion of the *smodels* program. Then we iteratively compute models of the completion using PB solvers. Whenever a non-stable model is found, we add to the completion loop formulas that guarantee that the same non-stable model will not be computed again. Our algorithm is shown in Figure 1.

Input: P — a ground logic program (possibly with weight atoms)
    A — a pseudo-boolean solver
Output: M — a stable model of P if A finds one; "Failed" otherwise

BEGIN
compute the completion $comp(P)$ of P represented in PB logic;
do
  M := a model of $comp(P)$ found by A;
  if (M does not exist) output "Failed" and terminate;
  if (M is stable) output M and terminate;
  compute the reduct P^M of P with respect to M;
  compute the greatest stable model M' under M in P^M;
  M^- := $M \setminus M'$;
  find all maximal loops in M^-;
  add loop formulas of the loops found in previous step to $comp(P)$;
while (**true**);
END

Fig. 1. Algorithm of *pbmodels*

3 *Pbmodels* — The Package

The *pbmodels* package, including executable PB solvers code, can be obtained at `http://www.cs.uky.edu/ai/pbmodels/pbmodels-0.1.tar.gz`. It is also installed at the *asparagus* site `http://asparagus.cs.uni-potsdam.de`.

The package contains the source code of `pbmodels` and supported PB solvers: *satzoo* [5], *pbs* [6], *wsatoip* [7], and *wsatcc* [8]. The first two are complete PB solvers while the last two are incomplete PB solvers based on *walksat* [9] local-search algorithm. In addition, the package contains also two scripts:

– `esrapl`: a perl script that recovers the structure of rules and weight atoms when `lparse` grounds the input program and converts it to a normal form required by *smodels*. We found that PB solvers are more effective when the original structure

of rules and weight atoms is restored. We designed `esrapl`[1] to "undo" conversion of ground programs to their normal form by `lparse`.

- `convert2`: a perl script that takes a theory in the logic PL^{wa}, and produces PB theories that are accepted by different PB solvers. It is invoked by the command:
  `convert2 <target format> <source file>`
  Values for `<target format>` are: `satzoo`, `pbs`, `wsatcc`, or `wsatoip`. They specify the format, to which the input PL^{wa} theory in the file `<source file>` is to be converted.
  The script can be used as a stand alone format translation program.

4 Input, Output, and How to Invoke *Pbmodels*

Pbmodels accepts on input programs obtained by grounding *smodels* programs with `lparse` and processing the result with `esrapl`. The output of *pbmodels* is similar to that of *smodels*. The string "False" indicates that there are no stable models for the input program[2]. Otherwise, the first stable model of the input will be printed on the screen.

The main options for `pbmodels` are:

1. `--engine <engine name>`
   It specifies which PB solver to use. Values for this option are: `satzoo`, `pbs`, `wsatcc`, and `wsatoip`
2. `--option <list of options to the solver>`
   It specifies options for a solver selected with the `--engine` option. Everything after `--option` is passed to the solver.

In addition, there are also options that allow the user to specify the path to executable programs if they are not installed to the default bin directory. We do not discuss them here due to space constraints.

The following examples show how to invoke *pbmodels*. We assume that the file *prog.lp* contains an input *smodels* program. We also assume that *satzoo* is to be used as the back-end engine.

```
lparse prog.lp | esrapl | pbmodels --engine satzoo
```

The following command will invoke *pbmodels*, use *satzoo*, and pass specific *satzoo* options to *satzoo*:

```
lparse prog.lp | esrapl | pbmodels --engine satzoo
--option -no-rand
```

5 Performance

We report here briefly on our experiments comparing the performance of *pbmodels* and *smodels*. We considered several benchmark problems. Due to lack of space, we present

[1] `lparse` spelled backwards.

[2] If an *incomplete* solver is used, "False" means that no stable models were found; the program may actually have stable models.

here only the results concerning the *traveling salesperson problem*, and the *weighted n-queens problem*. Specifications of both problems use general weight atoms. Additional experimental results can be found at `http://www.cs.uky.edu/ai/pbmodels`.

We tested *pbmodels* with the four *PB* solvers mentioned earlier and compared the results with those obtained for *smodels*. All experiments were run on machines with 3.2GHz Pentium 4 CPU, 1GB memory, running Linux with kernel version 2.6.11, gcc version 3.3.4.

In the following tables we report the number of instances a solver solved, and the number of times a solver won among all solvers, and among the complete solvers (that latter metric for complete solvers only). We also report the average and the median running times in seconds, over all instances that do not time out[3].

Table 1 shows results on the traveling salesperson problem. We randomly generated 50 weighted complete graphs containing 20 vertices. For each of them, we set the upper bound on the length of the TSP cycle to $w = 62$. With this bound, the solvers we tested find that 31 instances have solutions. For the remaining 19, none of the solvers was able to find a solution within the 3000-second time limit we set.

Table 1. TSP Problem

TSP	*# of Instances*	*# of Times Won*		*Timing*	
$(n = 20, w = 62)$	*Solved*	*v.s. All*	*v.s. Complete*	*Mean*	*Median*
smodels	19/50	1	7	1558.37	1637.14
pbmodels-satzoo	19/50	2	16	696.42	461.25
pbmodels-pbs	1/50	0	0	1482.24	1482.24
pbmodels-wsatcc	19/50	6	–	28.39	6.59
pbmodels-wsatoip	28/50	22	–	7.20	1.43

Among complete solvers, *pbmodels-satzoo* performs better than the other two. Even though *pbmodels-satzoo* and *smodels* solved the same number of instances, *pbmodels-satzoo* won more times among the three complete solvers. Moreover, the average running time of *pbmodels-satzoo* is about half of that of *smodels* and the median running time is about $1/3$ of that of *smodels*. Over all, *pbmodels-wsatoip* is the winner. It solves the largest number of instances. Furthermore, its average and median running times are about three orders of magnitude less than the running time of *smodels*.

Table 2 shows the results on the *weighted n-queens problem*. An instance to this problem consists of n^2 non-negative integer weights, one for each square of an $n \times n$ chessboard, and of an integer bound w. A solution to an instance is a placement of n queens on the chessboard so that they do not attack each other and the total weight of the placement (the sum of the weights of the squares occupied by queens) is no greater than w. We randomly generated 50 weighted "chessboards" of the size 20×20 and, in each case, we set $w = 50$. In this group, 29 instances have solutions. For the remaining 21 we do not know whether they have solutions or not; none of the solvers we tested could decide that within the set time limit.

[3] We do not include the time used by `lparse` and `esrapl` in the time reported, as we are only interested in the effectiveness of solvers.

Table 2. Weighted NQueens Problem

W-NQueens ($n = 20, w = 50$)	*# of Instances Solved*	*# of Times Won*		*Timing*	
		v.s. All	*v.s. Complete*	*Mean*	*Median*
smodels	2/50	0	2	697.81	661.20
pbmodels-satzoo	0/50	0	0	N/A	N/A
pbmodels-pbs	0/50	0	0	N/A	N/A
pbmodels-wsatcc	29/50	15	–	1.01	0.33
pbmodels-wsatoip	29/50	14	–	0.44	0.35

We observe that *pbmodels-wsatcc* and *pbmodels-wsatoip* outperformed *smodels*. Among the complete solvers, *smodels* is slightly better than *pbmodels-satzoo* and *pbmodels-pbs* (*smodels* managed to solve two instances in this category, other complete solvers timed-out on all). However, the local-search solvers are the overall winners in all metrics considered.

6 Conclusions

We have presented software package *pbmodels* that computes models of *smodels* programs. The key feature of our system is that it supports the use of off-the-shelf *PB* solvers developed by the satisfiability community and capable of process weight atoms directly.

Our experiments show that *pbmodels* performs better than *smodels* on benchmarks we considered. The results were especially good when local-search *PB* solvers were used. We observed the same phenomenon in experiments on other problems we considered: weighted Latin square problem, magic square problem, vertex-cover problem and the tower-of-Hanoi problem. The last two problems use only cardinality atoms. Hence we were able to include *cmodels* [3] in those two tests, too. The results showed that *PB* solvers are generally faster than *cmodels* on these two benchmark families.

Acknowledgments

We acknowledge the support of NSF grants IIS-0097278 and IIS-0325063.

References

1. Simons, P., Niemelä, I., Soininen, T.: Extending and implementing the stable model semantics. Artificial Intelligence **138** (2002) 181–234.
2. Lin, F., Zhao, Y.: ASSAT: Computing answer sets of a logic program by SAT solvers. In: Proceedings of AAAI-2002, AAAI Press (2002) 112–117.
3. Babovich, Y., Lifschitz, V.: *Cmodels package* (2002) `http://www.cs.utexas.edu/users/tag/cmodels.html`.
4. Liu, L., Truszczyński, M.: Properties of programs with monotone and convex constraints. In: Proceedings of AAAI-05, AAAI Press (2005).

5. Eén, N., Sörensson, N.: An extensible SAT solver. In: Proceedings of SAT-2003. Volume 2919 of LNCS., Springer (2003) 502–518.
6. Aloul, F., Ramani, A., Markov, I., Sakallah, K.: PBS: a backtrack-search pseudo-boolean solver and optimizer. In: Proceedings of SAT-2003. 346 – 353.
7. Walser, J.: Solving linear pseudo-boolean constraints with local search. In: Proceedings of AAAI-97, AAAI Press (1997) 269–274.
8. Liu, L., Truszczyński, M.: Local-search techniques in propositional logic extended with cardinality atoms. In: Proceedings of CP-2003. Volume 2833 of LNCS, Springer (2003) 495–509.
9. Selman, B., Kautz, H., Cohen, B.: Noise strategies for improving local search. In: Proceedings of AAAI-1994, AAAI Press (1994) 337–343.

KMONITOR – A Tool for Monitoring Plan Execution in Action Theories*

Thomas Eiter, Michael Fink, and Ján Senko

Vienna University of Technology, Institute of Information Systems, Vienna, Austria
{eiter, michael, jan}@kr.tuwien.ac.at

Abstract. We present a monitoring tool for plan execution in non-deterministic environments, which are described in an action language, based on non-monotonic logic programming. Thanks to it, deviations of concrete executions from expected ones can be detected, and diagnostic explanations in terms of unsuccessful action executions can be obtained. The latter may be exploited for execution recovery, and may help in rectifying an incoherent view of the planning domain.

1 Introduction

In a non-deterministic environment, an agent's plan for achieving a goal by taking a sequence of actions might fail, if some of the actions do not materialize as expected. For this reason, the plan execution might be monitored in order to detect an execution failure or potential problems at an early stage, from which the agent may then recover. Execution monitoring was considered for logical domain descriptions in Golog [7,8] and Flux [6], and for the action language $\mathcal{AL}$ in the APLAgent Manager [1,2]. To our knowledge it has not been considered for other KR action languages such as $\mathcal{C}$, or $\mathcal{K}$, and in particular for non-deterministic domains. In [4,3], a general monitoring approach for logic-based action languages with transition-based semantics is presented, in which it is checked from time to time whether the current state complies with a set $\mathcal{T}$ of trajectories which describe the expected executions of the plan. If a discrepancy is detected, then the execution is not on track and the agent might suitably reconsider it; in order to diagnose discrepancies, points of failure in the execution are computed, which informally explain discrepancies applying Occam's Razor by the latest action execution which might have resulted in a "bad" outcome. Such information is useful for execution recovery, e.g., if actions are undone [3], but also for checking whether the user's understanding of the domain is coherent with the formalization.

Example 1. As a running example, we consider here a variant of the well-known Blocks World domain, in which a block being moved may end up at a location different from the intended one, because the agent might not grip it properly. Suppose we have the blocks $\mathtt{a}, \mathtt{i}, \mathtt{p}, \mathtt{r}, \mathtt{s}, \mathtt{x}$, and the plan

$$P = \langle \mathtt{move}(\mathtt{r},\mathtt{x}), \mathtt{move}(\mathtt{i},\mathtt{s}), \mathtt{move}(\mathtt{r},\mathtt{i}), \mathtt{move}(\mathtt{p},\mathtt{x}), \mathtt{move}(\mathtt{a},\mathtt{r}), \mathtt{move}(\mathtt{p},\mathtt{a})\rangle$$

for reaching the goal state S_6 from the initial state S_0 in 6 steps as in Figure 1, which shows all trajectories for P that establish the goal. If now at stage 4 the discrepancy is

* Work supported by grants of FWF (P16536-N04) and the EC (FET-2001-37004 WASP).

C. Baral et al. (Eds.): LPNMR 2005, LNAI 3662, pp. 416–421, 2005.
© Springer-Verlag Berlin Heidelberg 2005

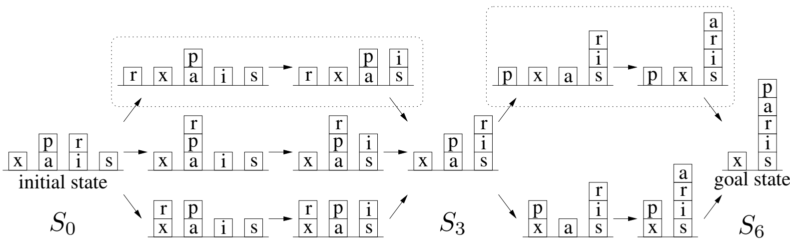


Fig. 1. Goal-establishing trajectories for the example plan

detected that p is on r, then the execution will fail, since the next action $\texttt{move}(\texttt{a}, \texttt{r})$ can not be taken. If all other blocks are situated as in S_3, an explanation is that the preceding action $\texttt{move}(\texttt{p}, \texttt{x})$ has failed (see [4, Ex. 3] for a formal description).

In general, not all goal-establishing trajectories might be equally desired, and some preferred over others. To model this, $\mathcal{T}$ contains all preferred trajectories.

Example 2. In our example, let the preferred trajectories $\mathcal{T}$ be those in which no block unintentionally falls on the table during execution. Hence, from the goal-establishing trajectories in Figure 1, those which pass through the dotted area are not preferred.

KMONITOR (*http://www.kr.tuwien.ac.at/research/monitoring*) implements the execution monitoring approach of [4,3] for the action language $\mathcal{K}$ on top of the DLV$^{\mathcal{K}}$ planning system [5]. Fig. 2 shows the main loop, which is entered when monitoring is issued. To keep the monitoring overhead low, the current state is analyzed only at certain *checkpoints*, which are determined by a respective component from a *checkpoint policy* specified by a non-monotonic logic program (see Section 2). State analysis is done by the tool KDIAGNOSE implementing the diagnosis method from [4]. If a discrepancy is detected, control is returned with this information and any diagnoses found.

Example 3. Suppose that blocks $\texttt{x}, \texttt{p}, \texttt{r}$ and $\texttt{s}$ are known to be heavy, and that the policy is to check each time when a heavy block was moved (as such moves bear high likelihood of failure). Then, the checkpoints would be the stages 1, 3, 4, and 6.

2 Checkpointing

Rather than checking for a discrepancy after each step, we may check only at certain stages, e.g., if a stage has higher likelihood of failure, or do a periodic check. Then monitoring can be less intrusive and the execution of the whole plan will be faster.

In order to select stages for a discrepancy check, a "checkpointing policy" is specified in terms of a logic program, which has facts and rules over the sets of fluents and actions from the domain. The policy is evaluated for all stages in the plan, and if a fact $\texttt{checkpoint}(t)$ is true for the current stage, then a check for discrepancy is issued.

We distinguish two types of checkpointing policies – static and dynamic ones. In the static case, checkpoints are calculated one and for all from the policy logic program

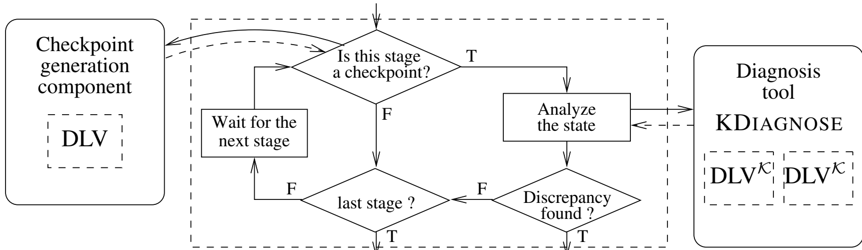


Fig. 2. Monitoring Loop in KMONITOR

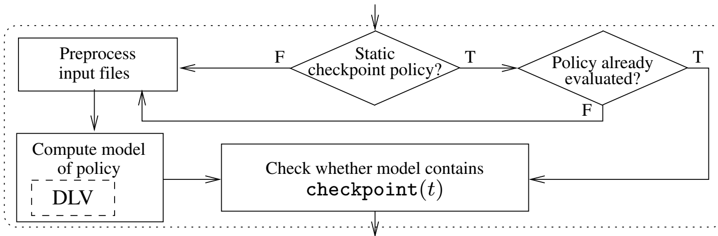


Fig. 3. Checkpoint generation component

and the plan. In the dynamic case, a part of the current state is taken into consideration, yielding more expressive power, but also the need for re-evaluating checkpoints. Figure 3 gives an overview of the checkpoint generation utility.

Static Checkpoints. When using a static checkpointing policy, we compute the model of a logic program, comprised of the checkpoint definition file and the modified plan. Actions of the plan are rewritten into facts with timestamps. (E.g., an action $\mathtt{move}(\mathtt{a}, \mathtt{r})$ occurring at stage 2 is rewritten into $\mathtt{move}(\mathtt{a}, \mathtt{r}, 2)$.) The checkpoint definition file contains rules over the modified action predicates and the special predicate $\mathtt{checkpoint}(t)$, whose truth value defines whether we have to do a check at stage t or not.

Example 4. To force a check after each move involving a heavy block, we can define the following static checkpointing policy:

$$\begin{array}{rl} \mathtt{heavy}(\mathtt{X}). & \text{for } \mathtt{X}{\in}\{\mathtt{x}, \mathtt{p}, \mathtt{r}, \mathtt{s}\}\text{, and} \\ \mathtt{checkpoint}(\mathtt{Tj}) \ :- & \mathtt{move}(\mathtt{X}, \mathtt{Y}, \mathtt{Ti}),\ \mathtt{heavy}(\mathtt{X}),\ \mathtt{Tj} = \mathtt{Ti} + 1. \end{array}$$

Dynamic Checkpoints. A dynamic checkpointing policy also involves information about some fluents of the current state; intuitively, they are sensed to steer the checkpointing. Therefore, we need to find models of the logic program – made up from the checkpoint declaration file, the plan, and partial state information – at each stage. At stage t we compute a model of this program, and if $\mathtt{checkpoint}(t)$ is true, a check for discrepancy is executed. The current stage t may be accessed via $\mathtt{now}(t)$.

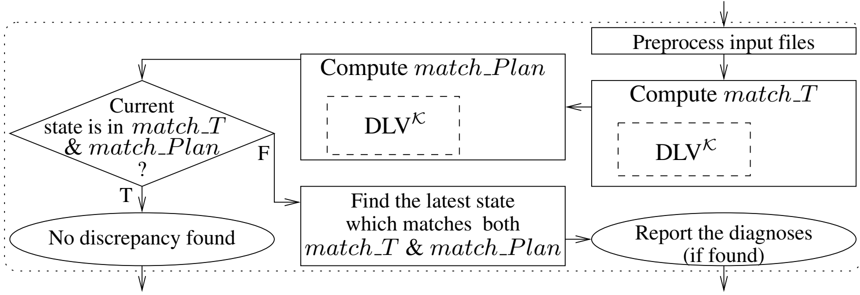


Fig. 4. KDIAGNOSE components and control flow

Example 5. For issuing a check after each step in which a block ends on a different location than intended, we can use the following dynamic checkpointing policy:

$$\texttt{checkpoint(Tj)} :- \texttt{move(X, Y, Ti)}, \texttt{not on(X, Y)}, \texttt{Tj} = \texttt{Ti} + 1.$$

This policy will be evaluated at each stage and its result depends on the current state.

In a "sleep" mode, policy re-evaluation can be suppressed until the next provisional checkpoint according to the last evaluation.

3 Diagnosis of Discrepancies

Before we detail KDIAGNOSE, we informally introduce some terminology (cf. [4]).

There is a *discrepancy* between a state S_i and a set of preferred trajectories $\mathcal{T}$ relative to a plan P, if there is no trajectory $S'_0, A_0, S'_1, \ldots, S'_{n-1}, A_{n-1}, S'_n$i.e., an alternating sequence of states and action occurrences in $\mathcal{T}$, such that $S'_i = S_i$ and S'_n is a goal state. Furthermore, we identify the point of failure for an observed discrepancy by finding the latest time stamp after which every evolution of S_i deviates from every trajectory in $\mathcal{T}$. Thus, a pair (S_k, k) is a *point of failure*, or a *diagnosis*, if:

(S1) Some evolution of S_i matches a goal-establishing trajectory in $\mathcal{T}$ at stage k ($0 \leq k < i \leq n$) in state S_k and deviates at stage $k + 1$.
(S2) No evolution of S_i matches a goal-establishing trajectory in $\mathcal{T}$ at stage $k' > k$.

Example 6. If, in our running example, (i) at stage 1 block r is on the table, or (ii) $\texttt{on(p, r)}$ holds at stage 4 while the remaining blocks are as depicted in S_3, then in both cases we cannot find a preferred trajectory with a corresponding state, i.e., we detect a discrepancy. For (i) still a feasible trajectory to the goal exists (stages 1 and 2 along the dotted area), not so for (ii). Thus, discrepancy (i) cannot be detected in, or after, stage 3. Observe also that $(S_3, 3)$ is the only diagnosis for (ii) and no diagnosis exists for (i).

KDIAGNOSE detects discrepancies and, if found, computes all diagnoses as follows (cf. Figure 3). Besides P, $\mathcal{T}$, and S_i it takes a domain description and the planning problem as inputs. In a preprocessing step, the current state and the planning problem are transformed into slightly modified planning problems for calculating

- the set $match_T$ of all goal-establishing preferred trajectories, and
- the set $match_Plan$ of all evolutions leading to S_i according to P.

Clearly, if S_i is on a goal-establishing preferred trajectory of $match_T$ (and thus also in $match_Plan$), then there is no discrepancy. Otherwise, KDIAGNOSE computes all diagnoses by comparing the above sets and searching maximal states S_j at which possible evolutions and goal-establishing preferred trajectories coincide.

4 Implementation

Our implementation of KMONITOR builds on KDIAGNOSE, which uses the DLV$^{\mathcal{K}}$ system to compute diagnoses for a given state. It invokes the diagnosis tool sequentially at all stages singled out by the checkpoint policy. The inputs to KMONITOR are:

- `plan` - a plan (in DLV$^{\mathcal{K}}$syntax) to monitor for discrepancies;
- `checkpoints.dl` - the checkpoint definition file;
- `background.dl` and `K.plan` - the domain description and planning problem;
- `T.plan` - a list of preferred trajectories, or alternatively a (modified) planning problem defining preferred trajectories (by its solutions);
- `state.*`, `cstate.*` - state & checkpointing information about the plan execution.

The checkpoint generation component is invoked as described in Section 2. Checkpoint computation is accomplished by computing models using the DLV system. For static policies, facts `checkpoint`(t) are extracted into a temporary file, and at the respective stages KDIAGNOSE is invoked for state analysis. A state t of a concrete partial execution (run) of the plan is fetched from file `state.`t. For simulation, a run may be automatically generated using DLV$^{\mathcal{K}}$(e.g., by randomly generating a trajectory for a modified planning problem). In the dynamic case, checkpointing state information is fetched from file `cstate.`t. If state information is missing, checkpointing is skipped.

Example 7. The static checkpoint policy of Ex. 4 yields `checkpoint`(t) , $t \in \{1, 3, 4, 6\}$. For a plan execution as in Ex. 6 (ii), KMONITOR reports a discrepancy for state S_4:

```
> No error at stage 1.
> No error at stage 3.
> Error at stage 4 - Point of failure at stage 3.
> State info: {on(x,table), on(a, table), on (p,a), ...
```

The dynamic policy of Ex. 5 would yield a run with the same result, but the check point policy would be evaluated at each step, and a check would occur only at stage 4. Note that if at stage 1 block r would unintentionally end on block p, then we would have another check but would not detect any discrepancy. We also remark that by simple refinements of our dynamic policy we could avoid evaluation at each step.

5 Conclusion and Future Work

KMONITOR is an execution monitoring tool for non-deterministic domains utilizing action language $\mathcal{K}$. It detects deviations from intended execution paths and computes explanations for discrepancies. By means of a checkpointing policy, discrepancy checking

and diagnosis can be restricted to certain stages of the execution. A detailed comparison with [1] is left for a longer version of the paper. For a comparison and further references to related work on diagnosis in answer-set programming, the reader is referred to [4].

Currently KDIAGNOSE handles no concurrent actions, which however is easy to overcome. Generalizing [4], it shall also support diagnosis w.r.t. partial state information restricted to a focus of interest. Finally, we plan to extend KMONITOR towards a system capable of giving recovery support as well. To this end, implementing and integrating techniques for recovery as described e.g. in [3] is envisioned.

References

1. Balduccini, M.: APLAgent Manager. krlab.cs.ttu.edu/~marcy/APLAgentMgr/index.html.
2. Balduccini, M., Gelfond, M.: Diagnostic reasoning with A-Prolog. Theory and Practice of Logic Programming, 3(4–5), 425–461. 2003.
3. Eiter, T., Erdem, E., Faber, W.: Plan Reversals for Recovery in Execution Monitoring. In *Proc. NMR*, 147–154. 2004.
4. Eiter, T., Erdem, E., Faber, W., Senko, J.: A logic-based approach to finding explanations for plan execution discrepancies. TR INFSYS RR-1843-04-03. 2004.
5. Eiter, T., Faber, W., Leone, N., Pfeifer, G., Polleres, A.: A logic programming approach to knowledge-state planning, II: The DLV$^{\mathcal{K}}$ system. AI, 144(1–2), 157–211. 2002.
6. Fichtner, M., Großmann, A., Thielscher, M.: Intelligent execution monitoring in dynamic environments. Fundamenta Informaticae, 57(2–4), 371–392. 2003.
7. Giacomo, G. D., Reiter, R., Soutchanski, M.: Execution monitoring of high-level robot programs. Proc. of KR'1998, 453–465. 1998.
8. Soutchanski, M.: High-level robot programming and program execution. Proc. of ICAPS Workshop on Plan Execution. 2003.

The *nomore++* System

Christian Anger, Martin Gebser, Thomas Linke,
André Neumann, and Torsten Schaub*

Institut für Informatik, Universität Potsdam, Postfach 90 03 27, D–14439 Potsdam

Abstract. We present a new answer set solver *nomore++*. Distinguishing features include its treatment of heads and bodies equitably as computational objects and a new hybrid lookahead. *nomore++* is close to being competitive with state-of-the-art answer set solvers, as demonstrated by selected experimental results.

1 Introduction

A large part of the success of Answer Set Programming (ASP) is owed to the easy availability of efficient solvers. We present a new ASP solver, called *nomore++* that pursues a hybrid approach in combining features from literal-based approaches, like *smodels* [1] and *dlv* [2], with the rule-based approach of its predecessor *noMoRe* [3]. To this end, it treats heads and bodies equitably as computational objects. We argue that this approach allows for more effective (in terms of search space pruning) choices than obtainable when dealing with either heads or bodies only. In particular, we demonstrate that the resulting hybrid lookahead operation allows for propagating more than previous approaches. Also, we detail a special strategy keeping assignments unfounded-free and empirically show that it outperforms *smodels* on relevant benchmarks. Another feature of *nomore++* is its configurable operator-based design. The system is available at [4].

2 Theoretical Background

The *nomore++* system deals with normal logic programs under the *answer set semantics* [5]. A *normal logic program* is a finite set of rules of the form $p_0 \leftarrow p_1, \ldots, p_m, not\ p_{m+1}, \ldots, not\ p_n$, where $n \geq m \geq 0$, and each p_i $(0 \leq i \leq n)$ is an *atom*. A *literal* is an atom p or its (default) negation $not\ p$. For such a rule r, let $head(r) = p_0$ be the *head* of r and $body(r) = \{p_1, \ldots, p_m, not\ p_{m+1}, \ldots, not\ p_n\}$ be the *body* of r. For a program Π, we write $head(\Pi) = \{head(r) \mid r \in \Pi\}$ and $body(\Pi) = \{body(r) \mid r \in \Pi\}$. Without loss of generality, we assume that each atom of a program is the head of at least one rule in the program.[1]

Given a normal logic program Π, *nomore++* computes the answer sets of Π. Unlike other solvers, such as *smodels* [1] and *dlv* [2],[2] the *nomore++* approach is based

* Affiliated with the School of Computing Science at Simon Fraser University, Burnaby, Canada.

[1] Atoms not occurring as heads are necessarily false. *nomore++* removes such atoms during preprocessing.

[2] Note that *dlv* is designed to handle disjunctive logic programs, which are on a higher complexity level than normal ones.

C. Baral et al. (Eds.): LPNMR 2005, LNAI 3662, pp. 422–426, 2005.
© Springer-Verlag Berlin Heidelberg 2005

on an extended concept of assignments. That is, *nomore*++ maps heads *and* bodies of a program into $\{\oplus, \ominus\}$, indicating whether a head or body is true or false, respectively. Given program Π, we define a (partial) assignment as a partial mapping $A : head(\Pi) \cup body(\Pi) \rightarrow \{\oplus, \ominus\}$. For simplicity, we often represent such assignments as pairs $(A^{\oplus}, A^{\ominus})$, where $A^{\oplus} = \{x \mid A(x) = \oplus\}$ and $A^{\ominus} = \{x \mid A(x) = \ominus\}$. Treating heads and bodies equitably as computational objects provides great flexibility. Bodies can be viewed as conjunctions, and their explicit representation allows for reasoning about applicability of rules, in addition to atoms' truth values. Structurally more complex objects have recently been deployed in SAT [6] and neighboring fields [7,8]. However, to the best of our knowledge this has not been done in ASP, so far.

nomore++ is a highly flexible, runtime-configurable system. Flexibility roots on an operator-based design, featuring (among others) the following basic operators: i) Forward propagation operator $\mathcal{P}$, ii) Backward propagation operator $\mathcal{B}$, iii) *Unfounded set* [9] operator $\mathcal{U}$, and iv) Choice operator $\mathcal{C}$.[3] Omitting details, $\mathcal{P}$ generalizes *Fitting's operator* [11] to bodies, combined operators $(\mathcal{PU})$ coincide with the *well-founded operators* [9], and $(\mathcal{PB})$ and $\mathcal{U}$ correspond to *smodels*' functions *atleast* and *atmost* [1]. Differences to atom- and rule-based approaches come up at *nomore*++'s more general choice operator $\mathcal{C}$. For instance, $\mathcal{C}$ can assign $\oplus$ to a body, enabling propagation to decide all the body's literals and heads. Such complex choices are not possible with assignments restricted to atoms.

Following [12], we characterize the process of answer set formation by a sequence of assignments. Based on the above operators, one possible strategy is to determine deterministic consequences with propagation operators $\mathcal{P}$, $\mathcal{B}$, and $\mathcal{U}$ and to apply choice operator $\mathcal{C}$ whenever a fix-point of propagation is not total. We abbreviate this strategy by $(\mathcal{PBU})^*\mathcal{C}$, where $(\mathcal{PBU})^*$ denotes the closure under operators $\mathcal{P}$, $\mathcal{B}$, and $\mathcal{U}$. The $\oplus$-assigned atoms in a total assignment, constructed by $(\mathcal{PBU})^*\mathcal{C}$, form an answer set.[4] However, other strategies for answer set formation can also be shown to be sound and complete. For instance, we can safely skip either backward or forward propagation, yielding strategies $(\mathcal{PU})^*\mathcal{C}$ and $(\mathcal{BU})^*\mathcal{C}$. Due to its configurable design, *nomore*++ can handle those different strategies as well.

An important concept, distinguishing answer set programming from propositional logic, is well-foundedness. It is thus desirable that no $\oplus$-assigned atom becomes unfounded later on. In fact, no atom in the positive part $A^{\oplus}$ of an assignment can become unfounded, if each atom in $A^{\oplus}$ is the head of a rule whose body is non-circularly justified in $A^{\oplus}$. We call such an assignment *unfounded-free*,[5] and *nomore*++ supports the computation of unfounded-free assignments by providing choice operator $\mathcal{D}$ as an alternative to $\mathcal{C}$. As opposed to $\mathcal{C}$, $\mathcal{D}$ is restricted to bodies whose positive preconditions are already present in $A^{\oplus}$ and where only negative literals are undecided. On the one hand, $\mathcal{D}$ restricts possible choices. But on the other hand, assignments are kept unfounded-free, which pays off on non-tight problems (cf. Section 4).

[3] For a detailed operational characterization of *nomore*++, please refer to [10].

[4] Note that we derive a contradiction when different values are to be assigned to some head or body. In this case, a total assignment cannot be constructed.

[5] A related notion for disjunctive programs is described in [13].

In addition to propagation operators $\mathcal{P}$, $\mathcal{B}$, and $\mathcal{U}$, lookahead strengthens propagation by conflict-driven assertions (*nomore++* provides operator $\mathcal{L}$ for this [10]). Answer set solvers like *smodels* and *dlv* apply lookahead to atoms only. In contrast to them, *nomore++* provides a *hybrid* lookahead considering both heads and bodies. In order to limit efforts to approximately the same amount as with lookahead on atoms, *nomore++*'s hybrid lookahead assigns $\oplus$ to bodies and $\ominus$ to atoms only. As shown in [10], this restriction does not decrease strength of propagation. Rather we demonstrate in Section 4 that hybrid lookahead can save exponentially many choices in comparison to lookahead applied to either atoms or bodies only.

3 System

The input language of *nomore++* is generated by the grounder lparse [14] from normal logic programs obeying the format "Logic Programs V1.0" as defined in [15].[6] A major feature of *nomore++* is that operators can be selected at runtime, enabling the use of a multitude of strategies (combinations of operators). Via command line option `-op`, the propagation and choice operators to be used can be determined. As lookahead allows for different degrees of propagation within, one can also determine which set of operators to use during lookahead via command line option `-laop`.

nomore++'s default strategy applies operators $\mathcal{P}$, $\mathcal{B}$, and $\mathcal{U}$ in usual propagation as well as in lookahead. Furthermore, $\mathcal{D}$ is the default choice operator. Note that operators $\mathcal{P}$, $\mathcal{U}$, and $\mathcal{D}$ keep a given assignment unfounded-free, which is not guaranteed for $\mathcal{B}$ and lookahead. At the implementation level, *nomore++* uses the additional truth value $\otimes$ for distinguishing between the unfounded-free part of an assignment and the part that must eventually be true but is not non-circularly justified, yet. The virtue of this is that the scope of unfounded set operator $\mathcal{U}$ can be restricted to $\otimes$-assigned and unassigned heads and bodies. In fact, $\mathcal{U}$ is implemented in a "lazy fashion" ignoring the $\oplus$-part of an assignment. The *dlv* system uses a similar feature, the truth value "must be true" [16].

Internally, *nomore++*'s primary data structure consists of a body-head dependency graph [17]. This is a very efficient structure, as it only stores each head-atom and each distinct body of a program, instead of each rule as most other ASP-solvers do. E.g., measuring over 241 randomly chosen ground programs in [15], the ratio of the number of distinct bodies over the number of rules is 0.41.

4 Selected Experimental Results

Due to space limitations, we confine our listed experiments to selected benchmarks illustrating the major features of *nomore++*. A complete evaluation, including further ASP solvers, like *assat* and *cmodels*, can be found at the ASP benchmarking site [15]. All tests were run on an AMD Athlon 1.4GHz PC with 512MB RAM. A memory limit of 256MB as well as a time limit of 900s were enforced. All results given in Figure 1 reflect the average of 10 runs.

Benchmarks 1-a to 1-d are taken from [4] and demonstrate the advantage of the hybrid lookahead strategy. For comparisons, we have in *nomore++* implemented body-based lookahead ("Body LaH") in addition to hybrid lookahead ("Hybrid LaH"). Values

[6] *nomore++* currently does not support *smodels*-style cardinality and weight constraints.

on the x-axis are a measurement for the size of the problem, please check [4] for details. Examples denoted with “Body-Plus” (Figures 1-a and 1-b) are better suited for a body-based lookahead. The *nomore++* version with body-based lookahead outperforms *smodels* on these. Examples “Head-Minus” (Figures 1-c and 1-d) can be solved optimally with a head-based lookahead. Consequently, we have *smodels* outperforming *nomore++* with body-based lookahead. Please note that *nomore++* with hybrid lookahead always performs similar to the better suited approach.

Benchmark 1-e demonstrates the advantage of *nomore++*’s strategy of keeping assignments unfounded-free. The figure reflects results obtained on classical Hamiltonian

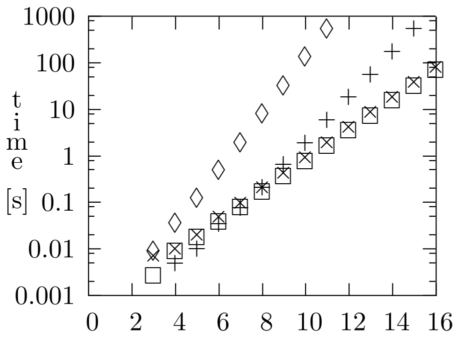


Fig. 1-a) Body-Plus, all AS

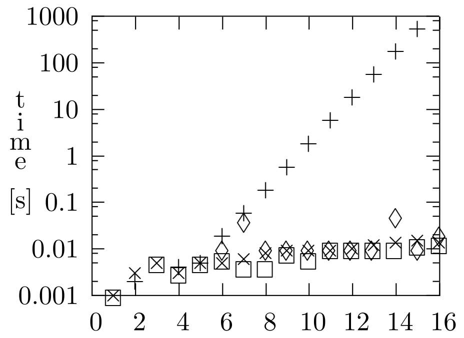


Fig. 1-b) Body-Plus, 1st AS

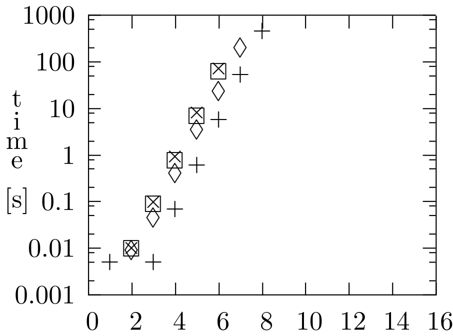


Fig. 1-c) Head-Minus, all AS

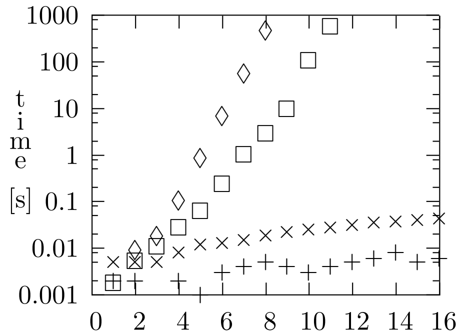


Fig. 1-d) Head-Minus, 1st AS

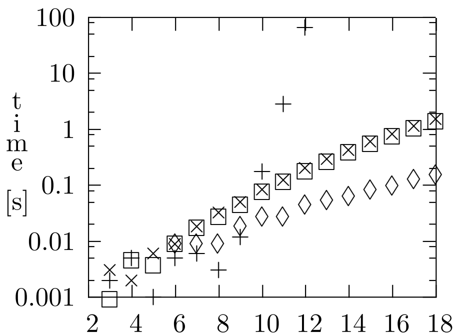


Fig. 1-e) Hamilton Cycles, 1st AS

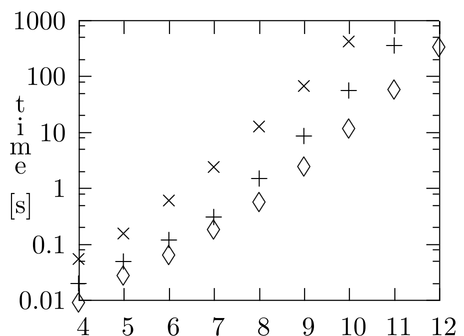


Fig. 1-f) EqTest, 1st AS

dlv ◊ *smodels* + *nomore++*(Body LaH) □ *nomore++*(Hybrid LaH) ×

cycle problems on complete graphs, where values on the x-axis reflect the number of nodes in the graph.

Let us note that, due to the fairly early development state of *nomore++*, its base speed is still inferior to more mature ASP solvers, like *smodels* or *dlv*. To reflect on this, we have in Figure 1-f included results from the "Equality Testing" benchmark taken from [15]. Please observe that, while *nomore++* performs worse than either *smodels* or *dlv*, it scales like the other two systems, indicating that only improvements with the implementation are needed.

Acknowledgments. This work was supported by DFG under grant SCHA 550/6-4 as well as the EC through IST-2001-37004 WASP project.

References

1. Simons, P., Niemelä, I., Soininen, T.: Extending and implementing the stable model semantics. Artificial Intelligence **138** (2002) 181–234
2. Leone, N., Faber, W., Pfeifer, G., Eiter, T., Gottlob, G., Koch, C., Mateis, C., Perri, S., Scarcello, F.: The DLV system for knowledge representation and reasoning. ACM Transactions on Computational Logic (2005) To appear.
3. Anger, C., Konczak, K., Linke, T.: `noMoRe`: A system for non-monotonic reasoning under answer set semantics. In Eiter, T., Faber, W., Truszczyński, M., eds.: Proceedings of the Int'l Conference on Logic Programming and Nonmonotonic Reasoning, Springer (2001) 406–410
4. (`http://www.cs.uni-potsdam.de/nomore`)
5. Gelfond, M., Lifschitz, V.: The stable model semantics for logic programming. In: Proceedings of the International Conference on Logic Programming, MIT Press (1988) 1070–1080
6. Järvisalo, M., Junttila, T., Niemelä, I.: Unrestricted vs restricted cut in a tableau method for Boolean circuits. Annals of Mathematics and Artificial Intelligence (to appear)
7. Baaz, M., Egly, U., Leitsch, A.: Normal form transformations. [8] chapter 5 273–333
8. Robinson, J., Voronkov, A., eds.: Handbook of Automated Reasoning. MIT Press (2001)
9. van Gelder, A., Ross, K., Schlipf, J.: The well-founded semantics for general logic programs. Journal of the ACM **38** (1991) 620–650
10. Anger, C., Gebser, M., Linke, T., Neumann, A., Schaub, T.: The nomore++ approach to answer set solving. Submitted for publication (2005)
11. Fitting, M.: Fixpoint semantics for logic programming: A survey. Theoretical Computer Science **278** (2002) 25–51
12. Konczak, K., Linke, T., Schaub, T.: Graphs and colorings for answer set programming. Theory and Practice of Logic Programming (2005) To appear.
13. Leone, N., Rullo, P., Scarcello, F.: Disjunctive stable models: Unfounded sets, fixpoint semantics, and computation. Information and Computation **135** (1997) 69–112
14. (`http://www.tcs.hut.fi/Software/smodels/`)
15. (`http://asparagus.cs.uni-potsdam.de`)
16. Faber, W., Leone, N., Pfeifer, G.: Pushing goal derivation in dlp computations. In: Proceedings of the International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'99). (1999) 177–191
17. Linke, T., Sarsakov, V.: Suitable graphs for answer set programming. In Baader, F., Voronkov, A., eds.: Proceedings of the International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR'05). Springer (2005) 154–168

SMODELS^A — A System for Computing Answer Sets of Logic Programs with Aggregates

Islam Elkabani, Enrico Pontelli, and Tran Cao Son

New Mexico State University, Las Cruces, NM 88003, USA
{tson, epontell, ielkaban}@cs.nmsu.edu

1 Introduction

In [2], we presented a system called $\mathbb{ASP}$-$\mathbb{CLP}$ for computing answer sets of logic programs with aggregates. The implementation of $\mathbb{ASP}$-$\mathbb{CLP}$ relies on the use of an external constraint solver (ECLiPSe) to deal with aggregate literals and requires some modifications to the answer set solver used in the experiment (SMODELS). In general, the system is capable of computing answer sets of arbitrary programs with aggregates, i.e., there is no syntactical restrictions imposed on the inputs to the system. This makes $\mathbb{ASP}$-$\mathbb{CLP}$ different from DLV^A (built BEN/5/23/04) [1], which deals with stratified programs only. $\mathbb{ASP}$-$\mathbb{CLP}$, however, is based on a semantics that does not guarantee minimality of answer sets. Furthermore, our experiments with $\mathbb{ASP}$-$\mathbb{CLP}$ indicate that the cost of communication between the constraint solver and the answer set solver proves to be significant in large instances.

In this work, we explore an alternative to $\mathbb{ASP}$-$\mathbb{CLP}$ and develop a new system for computing answer sets of logic programs with aggregates. We begin with the definition of a new semantics for programs with aggregates that has the following characteristics:

- It applies to *arbitrary* programs with aggregates, e.g., no syntactic restrictions on the use of aggregates, and it is as intuitive as the traditional answer set semantics.
- It *does not* explicitly require the satisfaction of desirable properties of answer sets (such as being *closed, supported,* or *minimal*), but the answer sets resulting from the new definition *naturally* satisfy such properties.
- It can handle aggregates as head of rules (not supported yet in our implementation).
- It can be implemented by integrating the definition directly in state-of-the-art answer set solvers. In particular, it requires only the addition of a module to determine the "solutions" of an aggregate, without any modifications to the mechanisms to computer answer sets.

The syntax of the language is similar to $\mathbb{ASP}$-$\mathbb{CLP}$—where a new type of literals (aggregate literals) is used; an aggregate literal has the form $F(\{X \mid p(X_1, \ldots, X_n)\})\, op\, Val$ where F is an aggregate function (e.g., SUM), and *op* is a relational operator (e.g., $=$, $\leq$). A similar literal with multisets is also available. The semantics and comparison with other approaches can be found in [3]. Its main features are:

- it defines the concept of *solution* of an aggregate ℓ as a pair $\langle X, Y \rangle$ such that every model M of the program satisfying $X \subseteq M$ and $Y \cap M = \emptyset$ also satisfies ℓ;
- it defines the *unfolding* of a program based on the notion of solution.

C. Baral et al. (Eds.): LPNMR 2005, LNAI 3662, pp. 427–431, 2005.
© Springer-Verlag Berlin Heidelberg 2005

The unfolding of a program with aggregates is a normal logic program whose answer sets can be computed using off-the-shelf systems. A set of atoms M is an answer set of a program with aggregates P iff it is an answer set of *unfolding*(P). We illustrate the semantics through the following examples.

Example 1. Let P_1 be the program

$$p(1). \qquad p(2). \qquad p(3) \leftarrow q. \qquad q \leftarrow sum(\{X \mid p(X)\}) > 5.$$

The only aggregate solution of $sum(\{X \mid p(X)\}) > 5$ is $\langle\{p(1), p(2), p(3)\}, \emptyset\rangle$ and *unfolding*(P_1) contains:

$$p(1). \qquad p(2). \qquad p(3) \leftarrow q. \qquad q \leftarrow p(1), p(2), p(3).$$

which has $M_1 = \{p(1), p(2)\}$ as its only answer set. M_1 is the only answer set of P_1.

Example 2. Consider the program P_2:

$$p(2). \qquad p(1) \leftarrow min(\{X \mid p(X)\}) \geq 2.$$

The aggregate literal $min(\{X \mid p(X)\}) \geq 2$ has a unique solution $\langle\{p(2)\}, \{p(1)\}\rangle$.

$$unfolding(P_2) = \{p(2). \qquad p(1) \leftarrow p(2), not\ p(1).\}$$

unfolding(P_2) does not have answer sets, i.e., P_2 does not have answer sets.

We will now describe SMODELS^A that implements the new semantics. Source code of the system can be found at `www.cs.nmsu.edu/~ielkaban/asp-aggr.html`.

2 The SMODELS^A System

Our main goal in developing SMODELS^A is to test the feasibility of a new approach to computing the answer sets of programs with aggregates by *(i)* computing the solutions of aggregate literals; *(ii)* computing the unfolding; and *(iii)* using standard answer set solvers to compute the answer sets. For this reason, we add to LPARSE and SMODELS two new modules. One for the preprocessing and another for the computation of the unfolding program. The overall structure of our system is shown in Fig. 1. The current implementation is built using SMODELS v.2.28 and LPARSE v.1.0.13.

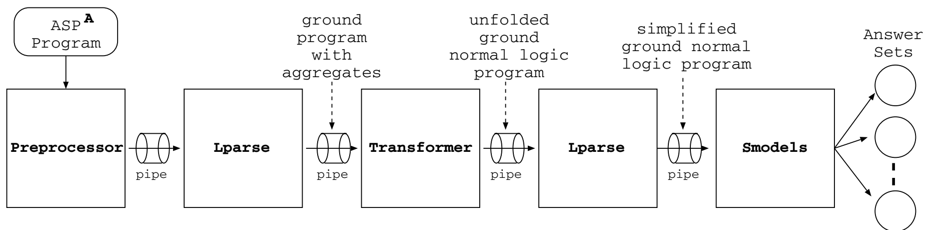


Fig. 1. Overall System Structure

Similar to the SMODELS system, the computation of answer sets of a program with aggregates is piped through several stages. In the 2nd and 4th stage, LPARSE is used. In the last stage, SMODELS is used. Let us detail the modules used in the other stages.

2.1 The Preprocessor

The *Preprocessor* is used to perform a number of simple syntactic transformations of the input program. These transformations are mostly aimed at rewriting the aggregate

literals in a format acceptable by LPARSE. An aggregate literal of the form $f(\{X \mid p(X,Y)\})$ op R is transformed into an atom, *t-aggregate atom*, of the form

```
''$agg''(f, ''$x'', p(''$x'', Y), R, op)
```

and a choice rule

```
{''$agg''(f, ''$x'', p(''$x'', Y), R, op)} ← type(Y)
```

where `type(Y)` is the domain predicate specifying the possible values of Y. For example, the rule

$$q \leftarrow sum(\{X \mid p(X)\}) > 3.$$

is transformed to:

```
q ← ''$agg''(sum,''$x'',p(''$x''),3,greater).
{''$agg''(sum,''$x'',p(''$x''),3,greater)}.
```

The resulting program is processed by LPARSE and by the *Transformer Module*.

2.2 The Transformer Module

The *Transformer Module* is the major component of SMODELS^A. It is responsible for the computing of the unfolding of the input programs and has four components: *Reader*, *Dependencies Analyzer*, *Aggregate Solver*, and *Rules Expander*. The overall organization of the *Transformer Module* is shown in Fig. 2. The *Transformer* is completely written in Prolog.

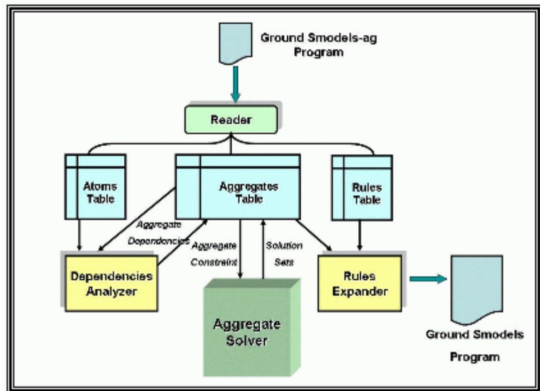


Fig. 2. Transformer Module

Reader. The *Reader* gets the output of the first LPARSE processing and constructs three tables: the *Atoms Table*, the *Rules Table*, and the *Aggregates Table*. These tables store the ground atoms, the ground rules, and the ground t-aggregate atoms (called aggregate atoms hereafter). For each aggregate atom, the *Reader* also stores other information, such as its aggregate function (e.g., SUM, COUNT, etc.), its relational operator (e.g., $>$, $<$, etc.), the compared value, the grouped variable, and the dependent atoms skeleton (e.g., $p(X)$ where X is the grouped variable). For example, the values for these attributes are SUM, $>$, 3, "$x", p("$x"), respectively, for the aggregate atom "$agg"(sum, "$x",p("$x"),3,greater).

Dependencies Analyzer (DA). The *DA* is responsible for the identification of the dependencies between aggregate atoms and atoms contributing to such aggregates. For each aggregate literal, the *DA* searches the *Atoms Table* for its atom dependencies, constructs a set (implemented as a list) of pointers to these atoms, and stores it as a part of the aggregate information in the *Aggregates Table*. These dependencies represent the domain from which the solutions of an aggregate constraint are built. For example, the set of

dependencies of the atom “$agg”(sum, “$x”,p(“$x”),3,greater) consists of all the atoms of the form $p(X)$ and is $\{p(1), p(2), p(3)\}$ in the previous example.

Aggregate Solver (AS). The main task of the *AS* is to compute a minimal solution set for each aggregate atom in the program. It contains several constraint solving procedures, one for each aggregate function. Presently, it supports SUM, AVG, MIN, MAX, and COUNT and the basics relational operators $>, <, \geq, \leq, =, \neq$. For every aggregate atom in the *Aggregates Table*, *AS* identifies its aggregate function and sends it, together with its set of dependencies, to the appropriate constraint solving procedure which produces either *(i)* a minimal set of solutions needed for the unfolding of the atom, if the aggregate literal has some solutions; or *(ii) false* otherwise. This information is then stored in the *Aggregates Table*. If we consider the previous example, *AS* will return the set $\{\langle\{p(1), p(2), p(3)\}, \emptyset\rangle\}$ for the aggregate atom “$agg”(sum, “$x”,p(“$x”),3,greater) with the set of dependencies $\{p(1), p(2), p(3)\}$. If the constant 3 in the aggregate literal is changed to 7, the *AS* will returns *false*.

Rules Expander (RE). The *RE* module completes the job of the *Transformer Module*, by computing the unfolding of the program. For each rule r in the *Rules Table*, it generates *unfolding*(r), the set of rules obtained from r by simultaneously replacing each aggregate literal in r by the unfolding of one of its solutions (stored in the *Aggregate Table*). The *RE* also simplifies the code to remove the temporary choice rules introduced by the *Reader*. *RE* also performs some optimizations, such as removing rules whose body contains an unsatisfiable aggregate literal. For the program P_1, the result of this step is the following program:

```
p(1).    p(2).    p(3) ← q.    q ← p(1),p(2),p(3).
```

Table 1. Benchmarks with Aggregates (times in sec.)

Program	Sample Size	SMODELS$^{\mathcal{A}}$ Time	Transformer Time	DLV$^{\mathcal{A}}$ Time
Company Control	20	0.010	0.080	N/A
Company Control	40	0.020	0.340	N/A
Company Control	80	0.030	2.850	N/A
Company Control	120	0.040	12.100	N/A
Shortest Path	20	0.220	0.740	N/A
Shortest Path	30	0.790	2.640	N/A
Shortest Path	50	3.510	13.400	N/A
Shortest Path (All Pairs)	20	6.020	35.400	N/A
Party Invitations	40	0.010	0.010	N/A
Party Invitations	80	0.020	0.030	N/A
Party Invitations	160	0.050	0.050	N/A
Seating	16/4/4	11.40	0.330	4.337
Employee Raise	15/5	0.57	0.140	2.750
Employee Raise	21/15	2.88	1.770	6.235
Employee Raise	24/20	3.13	2.420	26.50
NM1	125	0.11	0.10	N/A
NM1	150	0.16	0.13	N/A
NM2	125	1.44	0.80	N/A
NM2	150	2.08	1.28	N/A

3 Experiments and Benchmarks

We have experimented SMODELS^A with various benchmarks (some from the literature and some newly created) and compared it with DLV$^{\mathcal{A}}$ whenever possible (Table 1). The experiments have been performed on a Linux P4 (3.06GHz, 512MB). The column SMODELS^A reports the time for computing answer sets of the unfolded program, while **Transformer Time** reports the unfolding time. The performance results are acceptable in most cases; on stratified programs, our system is occasionally faster than DLV, and occasionally slower, depending on the type of aggregate (some have many solutions, that we precompute, and that are not required during answer set computation).

4 Discussion

We presented a new system for computing answer sets of logic programs with aggregates. The new system differs from our previous system in two ways: *(i)* it implements a different, intuitive, semantics, which leads only to minimal models; and *(ii)* it does not modify LPARSE and SMODELS. The result of our initial experimentation shows that this direction is promising. The system has not been optimized for performance and this will be our focus in the near future. In particular, we plan to

1. Improve the preprocessor, e.g., by using more sophisticated data structures (e.g., to speedup search of atoms during the *DA* phase) and to eliminate redundant aggregate atoms in the *Aggregate Table*.
2. Improve the aggregate solver to allow more than one grouping variable and additional aggregate functions (presently, it handles only one grouping variable and allows only basic aggregate functions);
3. Improve the rule expander to reduce the size of the unfolding program.

A more important work, that is in progress, is to extend our system to support a second characterization of our aggregate semantics, equivalent to the one mentioned here, which relies on unfolding w.r.t. a specific answer set; this is expected to reduce the size of the unfolding for many aggregates and simplifies the handling of aggregates in the head of the rules. Furthermore, from our experiments, it is obvious that there are aggregates that are better handled with the approach described in this paper (as they lead to a small unfolding), and others that would benefit from additional knowledge about the answer set we are building (i.e., delay the unfolding until the actual answer set computation). We plan to develop classification methods that will select the appropriate unfolding approach.

References

1. T. Dell'Armi, W. Faber, G. Ielpa, N. Leone, G. Pfeifer. Aggregate Functions in Disjunctive Logic Programming: Semantics,Complexity,and Implementation in DLV. *IJCAI*, 2003.
2. I. Elkabani, E. Pontelli, and T.C. Son. Smodels with clp and its applications: A simple and effective approach to aggregates in asp. *Int. Conference Logic Programming*, pp. 73–89, 2004.
3. T.C. Son, E. Pontelli, and I. Elkabani. A Translational Semantics for Aggregates in Logic Programming. Technical Report NMSU-CS-2005-005, New Mexico State University, 2005.

A DLP System with Object-Oriented Features

Francesco Ricca, Nicola Leone, Valerio De Bonis, Tina Dell'Armi,
Stefania Galizia, and Giovanni Grasso

Department of Mathematics, University of Calabria, Rende (CS) 87036, Italy

Abstract. The paper presents DLV$^+$a Disjunctive Logic Programming system with object-oriented constructs, including classes, objects, (multiple) inheritance, and types. DLV$^+$ is built on top of DLV (a state-of-the art DLP system), and provides a graphical user interface that allows to specify, update, browse, query, and reason on knowledge bases. Two strong points of the system are the powerful type-checking mechanism, and the advanced interface for visual querying.

1 Introduction

Disjunctive Logic Programming (DLP) is an advanced formalism for Knowledge Representation and commonsense Reasoning (KR&R)[1]. DLP is very expressive in a precise mathematical sense: it is able to express all problems belonging to the complexity class Σ_2^P. Moreover, the availability of a couple of efficient DLP systems, like DLV [2], GnT [3] and, more recently, the disjunctive version of Cmodels [4] make DLP a powerful tool for developing advanced knowledge-based applications [5,6].

The recent application of DLV in the emerging areas of Knowledge Management (KM) and Information Integration [8], has confirmed, from the one hand, the viability of the DLP exploitation. On the other hand, it has evidenced some limitations of DLP language and systems. As far as the language is concerned, the need to represent complex real-world entities, like classes, objects, compound objects, and taxonomies, has emerged [7]. Moreover, the DLP systems are missing the tools for supporting the programmers, like type-checkers and easy-to-use graphical environments, to manage the large and complex domains to be dealt with in real-world applications.

This paper describes the DLV$^+$ system, a first step to overcome the above limitations. It is a cross-platform development environment for knowledge modeling and advanced knowledge-based reasoning. The DLV$^+$ system allows to easily develop complex applications and allows to perform advanced reasoning tasks in a user friendly visual environment. The DLV$^+$ system seamlessly integrates the DLV system exploiting the power of a stable and efficient ASP solver.

A strong point of the system is its powerful language, extending DLP by object-oriented features. In particular, the language includes, besides the concept of **relations**, the object-oriented notions of **classes**, **objects** (class instances), **object-identity**, **complex-objects**, **(multiple) inheritance**, and the concept of modular programming by means of **reasoning modules**.

C. Baral et al. (Eds.): LPNMR 2005, LNAI 3662, pp. 432–436, 2005.
© Springer-Verlag Berlin Heidelberg 2005

A *class* can be thought of as a collection of individuals that belong together because they share some properties. An individual, or *object*, is any identifiable entity in the universe of discourse. Objects, also called class instances, are unambiguously identified by their object-identifier (oid) and belong to a class. A class is defined by a name (which is unique) and an ordered list of attributes, identifying the properties of its instances. Each attribute is identified by a name and can be of class type. This allows the specification of *complex objects* (objects made of other objects).

Classes can be organized in a specialization hierarchy (or data-type taxonomy) using the built-in *is-a* relation (*multiple inheritance*).

Relationships among objects are represented by means of *relations*, which, like classes, are defined by a (unique) name and an ordered list of attributes (with name and type). Importantly, DLP$^+$ supports two kind of relations, *base relations*, corresponding to basic facts (that can be stored in a database), and *derived relations* corresponding to facts that can be inferred by logic programs.

As in DLP, logic programs are a set of logic rules and constraints. However, DLP$^+$extends the definition of logic atom introducing class and relation predicates and complex terms, allowing a direct access to object properties. This way, DLP$^+$ rules merge, in a simple and natural way, the declarative style of logic programming with the navigational style of the object-oriented systems. In addition, DLP$^+$ logic programs are organized in *reasoning modules*, taking advantage of the benefits of modular programming.

Importantly, the strongly-typed nature of DLP$^+$ allowed the implementation of a number of **type-checking** routines that verify the correctness of a specification on the fly, resulting in a great help for the programmer.

Moreover, DLV$^+$ offers several important facilities driving the development of both the knowledge base and the reasoning modules. Using DLV$^+$, developers and domain experts can create, edit, navigate and query object-oriented knowledge bases by an easy-to-use **visual environment**, enriched by a full graphic **query interface** à la QBE.

2 Language Overview

Classes can be declared in DLV$^+$ by using the keyword `class` followed by the class name and by a comma separated list of attributes. Each attribute is a couple `(attribute-name : attribute-type)`. The attribute-type is either a user-defined class name, or a built-in class name (in order to deal with concrete data types, DLP$^+$ features two built-in classes *string* and *integer*). For instance, the class faculty with an argument of type string can be declared as follows:

```
class faculty(name:string).
```

Objects, that is class instances, are declared by asserting new facts. An instance for the class faculty, can be declared as follows:

```
#1:faculty(name:"Faculty of Science").
```

The string “Faculty of science” values the attribute name; while #1 is the *object-identifier (oid)* of this instance (each instance is equipped by a unique oid). The possibility to specify user-defined classes as attribute types allows for

complex objects, i.e. objects made of other objects. The following declaration of class *person* includes, besides *name* and *age*, an attribute of type *person*, namely, *partner*.

```
class person(name:string,age:integer,partner:person).
```

Note that this declaration is "recursive" (partner is of type person). A couple of partners can be specified as follows:

```
#2:person(name:"Max", age:30,partner:#3).
#3:person(name:"Mary", age:28, partner:#2).
```

Note that "Max" (identified by #2) is "Mary's" partner and vice versa ("Mary" is identified by #3). In general, the object identifier can be assigned explicitly by the programmer (any DLP constant can be used as oid); if omitted, the oid is automatically provided by the system. Moreover, arguments are identified by name, allowing for an easier way to access attributes.

Classes can be organized in a taxonomy. For example, the following *student* class extends the *person* class by two new attributes, *code* and *enrol*.

```
class student isa {person} (code:string,enrol:faculty).
```

Instances of the class student are declared as usual, by asserting new facts.

```
#6:student(name:"John",age:20,father:#2,partner:#7,code:0, enrol:#1).
#7:student(name:"Alice",age:20,father:#2,partner:#6,code:1, enrol:#1).
```

Like in common object-oriented languages , each instance of a sub-class becomes, automatically, an instance of all super classes (isa relation induces an inclusion relation between classes). In the example, "John" and "Alice" are instances of both *person* and *student*. Moreover, sub-classes inherit attributes from all super-classes. In the example, the student class has all attributes of the person class (inherited) plus the (local) attributes *code* and *enrol*.

The language provides a built-in most general class named *object* that is the class of all individuals and is a superclass of all DLP$^+$ classes.

Also multiple inheritance is supported. For example, class *stud_emp*, declared next, inherits from both class *student* and class *employee*.

```
class employee isa{person}(salary:integer,skill:string,company:string).
class stud_emp isa{student,employee}(workload:integer).
```

Attribute inheritance in DLP$^+$ follows the strategy adopted in the COMPLEX language, for a formal description refer to [9].

Relations represent relationships among objects. Base relations are declared like classes, and tuples are specified (as usual) asserting a set of facts (but tuples are not equipped with an oid). For instance, the base relation *colleague*, and a tuple asserting that "Max" and "Mary" are colleague, follows:

```
relation colleague(p1:person,p2:person).
colleague(p1:#2,p2:#3).
```

Classes and base relations are, from a data-base point of view, the extensional part of the DLP$^+$ language. Conversely, derived relation are the intensional (deductive) part of the language and are specified by using reasoning modules, which like DLP programs, are composed of logic rules and integrity constraints.

DLP$^+$ reasoning modules allow one to exploit the full power of DLP. As an example, consider the following module, encoding the team-building problem (compute a team satisfying some project restrictions).

```
class project(numEmp:integer, numSk:integer, budget:integer,
              maxSal:integer).

module(teamBuilding){
 inTeam(E,P) v outTeam(E,P) :- E:employee(), P:project().
 :- P:project(numEmp:N),not #count{E: inTeam(E,P)}=N.
 :- P:project(numSk:S),not #count{Sk: inTeam(employee(skill:Sk),P)}>=S.
 :- P:project(budget:B),not #sum{Sa,E: E:employee(salary:Sa),
                                  inTeam(E,P)}<=B.
 :- P:project(maxSal:M),not #max{Sa: inTeam(employee(salary:Sa),P)}<=M. }
```

Predicate arguments can be valued both by specifying simple terms like variables or object identifiers, and by using a nested class predicate (complex term) which works like a function (e.g., above, in the second and fourth constraint, the first argument of *inTeam* relation is an employee specified by a complex term). Note that, the complex predicate terms allow to combine in a simple and natural way, the declarative style of logic programming with the navigational one of the object-oriented systems.

Finally, in order to check the consistency of a knowledge base the user can specify global integrity constraints called axioms. Axioms can be defined as common logic rules by using the "::-" operator instead of ":-". Importantly, axioms are different by logic rules because model sentences that must be always true and do not derive new knowledge. For example the following axiom asserts that colleguaes must work at the same company.

```
X2:employee(company:C) ::- colleague(X1,X2), X1:employee(company:C).
```

3 System Architecture

The system architecture, depicted in Figure 1 (a), is a collection of seven modules: Parser, Data Handler, Type Checker, Intelligent Rewriter, Output Handler, Message Handler, and GUI.

The Parser has the job to analyze and load the content of a DLP$^+$ text file in the data structures supplied by the Data Handler. The Data Handler

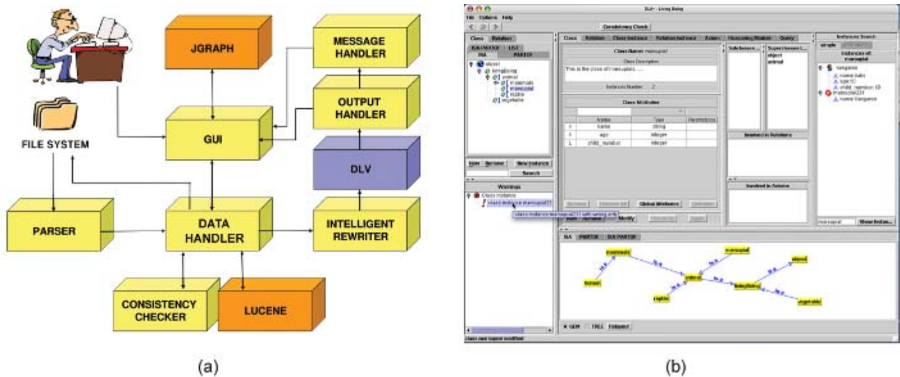


Fig. 1. The DLV$^+$architecture (a) and GUI (b)

provides all the methods needed to access and manipulate the knowledge base components. In particular, data indexing and full-text search are based on the open-source library Lucene. The Type Checker module implements a number of type checking routines, in order to ensure the consistency of a knowledge base. The Intelligent Rewriter module translates DLP^{+} knowledge base, reasoning modules and queries in an equivalent disjunctive logic program which runs on the DLV system. The Intelligent Rewriter features a number of optimization and caching techniques in order to reduce the time wasted interacting with the computational engine. Reasoning results and possible error messages are handled respectively by the Output Handler and by the Message Handler modules.

The user exploits the system through an easy-to-use visual environment called GUI (Graphical User Interface). The GUI combines a number of specialized visual tools for authoring, browsing and querying a DLP^{+} knowledge base. In particular, the GUI features a graph-based knowledge base viewer based on the JGraph library.

The GUI, shown in Figure 1 (b), is written in Java, while, the computational engine, (the DLV system) is a highly portable software, available for various platforms. Thus, the DLV^{+} system runs under a variety of operating systems.

An important feature of the system is the querying interface which provides both textual editing mode and visual editing mode à la QBE. The user can create queries without wondering about the syntax, simply selecting classes and relations and creating links between class attributes and relation parameters. Query results are presented to the user in a friendly way, while details about the interaction with DLV are hidden by the system.

References

1. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. New Generation Computing **9** (1991) 365–385.
2. Leone, N., Pfeifer, G., Faber, W., Eiter, T., Gottlob, G., Perri, S., Scarcello, F.: The DLV System for Knowledge Representation and Reasoning. ACM Transactions on Computational Logic (2005).
3. Janhunen, T., Niemelä, I.: GNT - A Solver for Disjunctive Logic Programs. In: Proc. of LPNMR 2004. LNCS, Fort Lauderdale, FL, USA, Springer (2004) 331–335
4. Lierler, Y.: Cmodels for Tight Disjunctive Logic Programs. In: Proc. of W(C)LP, Ulmer Informatik-Berichte, Ulm, Germany, Universität Ulm (2005) 163–166
5. Baral, C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press (2002)
6. Lobo, J., Minker, J., Rajasekar, A.: Foundations of Disjunctive Logic Programming. The MIT Press, Cambridge, Massachusetts (1992)
7. Massacci, F.: Computer Aided Security Requirements Engineering with ASP Nonmonotonic Reasoning, ASP and Constraints, Seminar N 05171, Dagstuhl, 2005.
8. Leone, N., T., Eiter, R., Rosati, etal: The INFOMIX System for Advanced Integration of Incomplete and Inconsistent Data. In: ACM SIGMOD 2005.
9. Greco, S., Leone, N., Rullo, P.: COMPLEX: An Object-Oriented Logic Programming System. IEEE Transactions on Knowledge and Data Engineering **4** (1992)

Testing Strong Equivalence of Datalog Programs - Implementation and Examples*

Thomas Eiter[1], Wolfgang Faber[2,**], and Patrick Traxler[1]

[1] Institute of Information Systems, Vienna University of Technology, 1040 Vienna, Austria
eiter@kr.tuwien.ac.at, e0027287@student.tuwien.ac.at
[2] Department of Mathematics, University of Calabria, 87030 Rende (CS), Italy
faber@mat.unical.it

Abstract. In this work we describe a system for determining *strong equivalence* of disjunctive *non-ground* datalog programs under the stable model semantics. The problem is tackled by reducing it to the unsatisfiability problem of first-order formulas in the Bernays-Schönfinkel fragment. We then employ a tableaux-based theorem prover, which (unlike most other currently available provers) is guaranteed to terminate for these formulas. To the best of our knowledge, this is the first strong equivalence tester for disjunctive non-ground datalog.

1 Introduction

Answer Set Programming (ASP) [1] is by now an acknowledged tool for knowledge representation and reasoning. The availability of efficient solvers has furthermore stimulated its use in practical applications in recent years. This development had quite some implications on ASP research. For example, increasingly large applications require features for modular programming. Another requirement is the fact that in applications, ASP code is often generated automatically by so-called frontends, calling for optimization methods which remove redundancies, as also found in database query optimizers. For these purposes, the more recently suggested notion of strong equivalence for programs [2,3] can be used. Indeed, if two ASP programs are strongly equivalent, they can be used interchangeably in any context. This gives a handle on showing the equivalence of ASP modules. If a program is strongly equivalent to a subprogram of itself, then one can always use the subprogram instead of the original program, a technique which serves as an effective optimization method.

So far, work on strong equivalence has mostly focused on propositional, or variable-free programs. The complexity of deciding whether two variable-free datalog programs are strong equivalent is in co-NP [4], however, when admitting variables, we obtain completeness for co-NEXPTIME [5]. Several systems have been proposed for testing strong equivalence of variable-free programs, some of which encode the problem again in ASP (e.g. [6]) or in propositional satisfiability [7,4].

In this work, we build on [4] and use a variant of the reduction described there, which in the non-ground case produces first-order formulas in the Bernays-Schönfinkel class which are unsatisfiable iff the original logic programs are strongly equivalent.

* This work was supported by FWF under project P18019-N04 and the European Commission under projects IST-2001-37004 WASP and IST-2001-33570 INFOMIX.
** Funded by an APART grant of the Austrian Academy of Sciences.

C. Baral et al. (Eds.): LPNMR 2005, LNAI 3662, pp. 437–441, 2005.
© Springer-Verlag Berlin Heidelberg 2005

2 Preliminaries

Disjunctive Datalog Programs. A *(disjunctive) rule* r is a formula

$$a_1(\bar{x}_1) \vee \cdots \vee a_n(\bar{x}_n) \text{ :- } b_1(\bar{y}_1), \cdots, b_k(\bar{y}_k), \texttt{not}\ b_{k+1}(\bar{y}_{k+1}), \cdots, \texttt{not}\ b_m(\bar{y}_m). \quad (1)$$

$n \geq 0$, $m \geq k \geq 0$, where all $a_i(\bar{x}_i)$ and $b_j(\bar{y}_j)$ are function-free atoms; if $n = 0$, r is also called a *constraint*. A *disjunctive datalog program* $\mathcal{P}$ is a finite set of rules and constraints. Two programs Π_1 and Π_2 are *strongly equivalent* [2] iff every program extensions $\Pi_1 \cup R$ and $\Pi_2 \cup R$ have the same answer sets [1].

Bernays-Schönfinkel Fragment of First-Order Logic. Any first-order sentence ψ of form

$$\exists x_1 ... x_k \forall y_1 ... y_l \varphi(x_1, ..., x_k, y_1, ..., y_l) \quad (2)$$

where φ is quantifier-free and without function and constant symbols, is a *Bernays-Schönfinkel formula*. Deciding satisfiability of such formulas is NEXPTIME-complete.

3 Reduction

In this section, we describe a reduction from the complementary problem of strong equivalence to satisfiability of Bernays-Schönfinkel formulas (whose quantifier-free part is in CNF), which is similar to the reduction defined in [4].

Given two logic programs Π and Π', let for each predicate p occurring in $\Pi \cup \Pi'$, be p' a fresh predicate of the same arity. Then

$$\Sigma(\bar{x}) := \bigwedge_{p \in Pred(\Pi \cup \Pi')} (p'(\bar{x}) \vee \neg p(\bar{x}))$$

For any rule r of the form (1), we define γ_r as the formula ($\bar{z} = \bar{x}_1 \cdots \bar{x}_n \bar{y}_1 \cdots \bar{y}_m$):

$$\forall \bar{z} \begin{pmatrix} (a_1(\bar{x}_1) \vee \cdots \vee a_n(\bar{x}_n) \vee b'_{k+1}(\bar{y}_{k+1}) \vee \cdots \vee b'_m(\bar{y}_m) \vee \neg b_1(\bar{y}_1) \vee \cdots \vee \neg b_k(\bar{y}_k)) \\ \bigwedge \\ (a'_1(\bar{x}_1) \vee \cdots \vee a'_n(\bar{x}_n) \vee b'_{k+1}(\bar{y}_{k+1}) \vee \cdots \vee b'_m(\bar{y}_m) \vee \neg b'_1(\bar{y}_1) \vee \cdots \vee \neg b'_k(\bar{y}_k)) \end{pmatrix}$$

For a program Π, we then define $\Gamma_\Pi := \bigwedge_{r \in \Pi} \gamma_r$, which we can easily rewrite to $\forall \bar{x} W_\Pi(\bar{x})$ where $W_\Pi(\bar{x})$ is a quantifier-free CNF. We next define a formula encoding the unique name assumption for the constants $\bar{c} = c_1, ..., c_n$ occurring in Π and Π':

$$U := \bigwedge_{i=1}^{n} (U_i(c_i) \wedge \bigwedge_{j \in \{1,...,n\} \setminus \{i\}} \neg U_i(c_j)).$$

For a formula φ, let φ^x_y be the formula with y replaced by x, and for a set S of formulas, let $S^x_y = \{\varphi^x_y \mid \varphi \in S\}$. As shown in [4], Π and Π' are *not* strongly equivalent iff at least one of the following two Bernays-Schönfinkel sentences is finitely satisfiable:

- $\exists \bar{u} \exists \bar{y} \forall \bar{z} \forall \bar{x} (U^{\bar{u}}_{\bar{c}}(\bar{u}) \wedge \Sigma(\bar{z}) \wedge W_\Pi{}^{\bar{u}}_{\bar{c}}(\bar{x}, \bar{u}) \wedge \neg W_{\Pi'}{}^{\bar{u}}_{\bar{c}}(\bar{y}, \bar{u}))$, resp.
- $\exists \bar{u} \exists \bar{y} \forall \bar{z} \forall \bar{x} (U^{\bar{u}}_{\bar{c}}(\bar{u}) \wedge \Sigma(\bar{z}) \wedge W_{\Pi'}{}^{\bar{u}}_{\bar{c}}(\bar{x}, \bar{u}) \wedge \neg W_\Pi{}^{\bar{u}}_{\bar{c}}(\bar{y}, \bar{u}))$.

(By the finite model property of Bernays-Schönfinkel, this is tantamount to unrestricted satisfiability.) Note that U, Σ, W_Π, and $W_{\Pi'}$ are CNFs, while $\neg W_\Pi$ and

$\neg W_{\Pi'}$ are not (moving negation inside, they are in DNF). Instead of the simple conversion to CNF, which is exponential in the worst case, we may for our purpose replace them with CNFs W_{Π}^* and $W_{\Pi'}^*$, respectively, which are equivalent with respect to satisfiability. To this end, we use the following transformation of a quantifier-free DNF $D(\bar{x}) = \bigvee_{i=1}^{n} \tau_i(\bar{x}_i)$ with free variables $\bar{x} = \bar{x}_1 \cdots \bar{x}_n$, where $\tau_i(\bar{x}_i) = l_{i,1}(\bar{x}_{i,1}) \wedge \cdots \wedge l_{i,m_i}(\bar{x}_{i,m_i})$ into a CNF $D^*(\bar{x})$ which is satisfiability-equivalent if the $\bar{x}_i$ and $\bar{x}_j$ are pairwise disjoint:

$$D^*(\bar{x}) = (s(d_1) \vee \cdots \vee s(d_n)) \wedge \bigwedge_{1 \leq i \leq n} \bigwedge_{1 \leq j \leq m_i} (l_{i,j}(\bar{x}_{i,j}) \vee \neg s(d_j))$$

where s is a new unary predicate symbol and $d_1, \ldots, d_n$ are fresh constant symbols.

Lemma 1. *$\forall \bar{x} D(\bar{x})$ is satisfiable iff $\forall \bar{x} D^*(\bar{x})$ is satisfiable, if all $\bar{x}_i$ and $\bar{x}_j$ are disjoint.*

We note that the size of $(\neg W_{\Pi})^*$ is linear in the size of Π, since it is linear in the size of W_{Π}, which in turn is linear in the size of Π. Let n_r and n_c be the number of predicate and constant symbols, respectively, in Π and Π'. Then, the size of Σ is linear in n_r and the size of U is quadratic in n_c.

Let $\Delta(\Pi, \Pi')$ denote the *clausal form* of $\exists \bar{u} \exists \bar{y} \forall \bar{z} \forall \bar{x} (U_{\bar{c}}^{\bar{u}}(\bar{u}) \wedge \Sigma(\bar{z}) \wedge W_{\Pi} \, {}_{\bar{c}}^{\bar{u}}(\bar{x}, \bar{u}) \wedge (\neg W_{\Pi'}(\bar{y}) \, {}_{\bar{c}}^{\bar{u}}(\bar{y}, \bar{u}))^*)$ after Skolemization, i.e., the set of clauses in $(U \wedge \Sigma(\bar{z}) \wedge W_{\Pi}(\bar{x}) \wedge (\neg W_{\Pi'}(f\bar{y}, f\bar{u}))^*)$. It can be easily generated, and its size is bounded as follows.

Proposition 1. *$|\Delta(\Pi, \Pi')| \leq k \cdot (|\Pi| + |\Pi'| + n_r + n_c^2)$ for some constant k.*

4 Implementation

The input language is similar to the one of DLV, but add-ons like built-ins, aggregates, weak constraints etc. are not supported. Also comments and anonymous variables are currently unsupported, as well as strong negation.[3]

The implementation is in `C++` employing a `flex/bison`-generated parser. Our basic data structures include a symbol table and a collection of rules. The generation of the clausal forms $\Delta(\Pi, \Pi')$ and $\Delta(\Pi', \Pi)$ is carried out via suitable functions working on these basic structures. We use the DARWIN theorem prover [4] as a back-end to solve the formulas. A distinguishing feature of DARWIN is that it is refutation-complete on our types of formulas, and thus strong equivalence of programs Π and Π', tantamount to refutations of $\Delta(\Pi, \Pi')$ and $\Delta(\Pi', \Pi)$, is definitely answered in all cases. Indeed, we are not aware of other provers which would guarantee this property.

The tool (including some examples) is available at `http://www.kr.tuwien.ac.at/students/prak_setest/`.

5 Examples

Example 1. Consider the program

```
Π: a(k1). a(k2).
   h(X):- a(X).  t(X):- h(X).  a(X):- t(X).   a(X):- h(X).
```

[3] In fact, strong negation $\neg a(\bar{x})$ is realized in DLV and other systems viewing $\neg a$ as a new predicate and adding a constraint $:\!-a(\bar{x}), \neg a(\bar{x})$; this can be respected and handled accordingly.

[4] http://goedel.cs.uiowa.edu/Darwin/

Π states that $a \subseteq h \subseteq t \subseteq a$, i.e. $a = h = t$. By dropping the last rule, we obtain
Π': `a(k1). a(k2). h(X):- a(X). t(X):- h(X). a(X):- t(X).`

The components of the formula $\Delta(\Pi, \Pi')$ are, in Darwin syntax, as follows.

Σ: `a_(X1):-a(X1). t_(X1):-t(X1). h_(X1):-h(X1).`

W_Π:
```
a(k1). a(k2).   a_(k1). a_(k2).
h(X):- a(X).   h_(X):- a_(X).   t(X):- h(X).   t_(X):- h_(X).
a(X):- t(X).   a_(X):- t_(X).   a(X):- h(X).   a_(X):- h_(X).
```

$W^*_{\Pi'}$:
```
-a(k1):-s__(1).      -a_(k1):- s__(6).    -a(k2):-s__(2).     -a_(k2):-s__(7).
-h(sk_1):-s__(3).    -h_(sk_4):-s__(8).   a(sk_1):-s__(3).    a_(sk_4):-s__(8).
-t(sk_2):-s__(4).    -t_(sk_5):-s__(9).   h(sk_2):-s__(4).    h_(sk_5):-s__(9).
-a(sk_3):-s__(5).    -a_(sk_6):-s__(0).   t(sk_3):-s__(5).    t_(sk_6):-s__(0).
s__(1), s__(2), s__(3), s__(4), s__(5), s__(6), s__(7), s__(8), s__(9), s__(0).
```

U: `u1(k1). -u1(k2). -u2(k1). u2(k2).`

A refutation is found by Darwin for $\Delta(\Pi, \Pi')$, and also for $\Delta(\Pi', \Pi)$. Hence, Π' and Π are strongly equivalent.

Example 2. Consider the two programs

Π:
```
t(X,Y):-a(X,Y).
t(X,Z):-t(X,Y),t(Y,Z).
```

Π':
```
t(X,Y):-a(X,Y).
t(X,Z):-a(X,Y),t(Y,Z).
```

which both compute the transitive closure of `a`. They are, however, not strongly equivalent, since $\Pi \cup \{t(1,2)., t(2,3).\}$ and $\Pi' \cup \{t(1,2)., t(2,3).\}$ have different answer sets. $\Delta(\Pi, \Pi')$ is unsatisfiable and $\Delta(\Pi', \Pi)$ is satisfiable, reflecting this fact.

6 Benchmarks

When experimenting with our tool, we have found that it often terminates quickly (less than one second), for instance for the examples presented in the previous section or for pairs of programs which differ substantially. We have been looking for parametric benchmark examples which create formulas that are increasingly hard to solve. These examples should be (1) scalable and (2) sufficiently similar to each other. We find that the following example interesting in this respect:

Example 3. (n-Layer TC Programs) Let Π_n be the following n-layer transitive closure:

```
t1(X,Y):- r(X,Y).      t1(X,Y):- r(X,Z), t1(Z,Y).
t2(X,Y):- t1(X,Y).     t2(X,Y):- t1(X,Z), t2(Z,Y).
   ...
tn(X,Y):- tn−1(X,Y).   tn(X,Y):- tn−1(X,Z), tn(Z,Y).
```

Π'_n is similar but with one additional redundant rule for each layer except the first:

```
t1(X,Y):- r(X,Y).       t1(X,Y):- r(X,Z), t1(Z,Y).
t2(X,Y):- t1(X,Y).      t2(X,Y):- t1(X,Z), t2(Z,Y).
t2(X,Y):- r(X,Z), t2(Z,Y).
   ...
tn(X,Y):- tn−1(X,Y).    tn(X,Y):- tn−1(X,Z), tn(Z,Y).
tn(X,Y):- r(X,Z), tn(Z,Y).
```

Table 1. Run-times for n-layer transitive closure

n	10	20	30	40	50	60
CPU time (sec)	0.38	1.43	3.66	7.93	13.55	23.56

The programs Π_n and Π'_n are strongly equivalent. We have tested this for various n on an 800MHz PowerPC with 1GB main memory, running GNU/Linux and DARWIN in version 08-27-04. The results are shown in Table 1. We can observe that the runtimes roughly double when increasing n by 10. Thus the scaling shows exponential behavior. On the other hand, viewed from computational complexity the n-layer TC is not among the "hard" instances of the problem; such hard instances could be systematically generated from complexity proofs.

We have also considered variants of these programs, where we added the rule

```
in(X,Y) ∨ out(X,Y) :- tn.
```

to Π_n, arriving at $\Pi_n^{final,\vee}$. To Π_n, we added the two body-shift variants of this rule

```
in(X,Y):-tn(X,Y), not out(X,Y).  out(X,Y):-tn(X,Y), not in(X,Y).
```

to obtain $\Pi_n^{final,\neg}$. The programs $\Pi_n^{final,\vee}$ and $\Pi_n^{final,\neg}$ are not strongly equivalent, and our tool was always very fast (less than one second) to decide this. Finding testcases which are not strongly equivalent and hard for our tool remains as an open issue.

7 Conclusion

We have implemented a non-ground strong equivalence tester, which works by a reduction to unsatisfiability of Bernays-Schönfinkel formulas, which are solved by the theorem prover Darwin, which is guaranteed to terminate on these formulas. The size of the resulting Darwin programs is nearly linear in the size of the input programs. Hence, the overall performance of testing strong equivalence depends heavily on the automated theorem prover. We have made a positive experience with our tool. We could find a class of problems which is apparently hard for the employed prover. With similar examples that are not strongly equivalent, its performance was, however, very good.

References

1. Gelfond, M., Lifschitz, V.: Classical Negation in Logic Programs and Disjunctive Databases. New Generation Computing **9** (1991) 365–385
2. Lifschitz, V., Pearce, D., Valverde, A.: Strongly Equivalent Logic Programs. ACM Transactions on Computational Logic **2** (2001) 526–541
3. Turner, H.: Strong Equivalence Made Easy: Nested Expressions and Weight Constraints. Theory and Practice of Logic Programming **3** (2003) 602–622
4. Lin, F.: Reducing Strong Equivalence of Logic Programs to Entailment in Classical Propositional Logic. In: Proc. KR-2002. 170–176
5. Eiter, T., Faber, W., Greco, G., Fink, M., Lembo, D., Tompits, H., Woltran, S.: Methods and Techniques for Query Optimization. TR D5.3, EC Project IST-2001-33570 (INFOMIX) (2004) Available at `http://www.mat.unical.it/infomix/`.
6. Eiter, T., Fink, M., Tompits, H., Woltran, S.: Simplifying logic programs under uniform and strong equivalence. In Proc. LPNMR-7, Springer (2004) 87–99
7. Pearce, D., Tompits, H., Woltran, S.: Encodings for Equilibrium Logic and Logic Programs with Nested Expressions. In Proc. EPIA 2001. 306–320

SELP - A System for Studying Strong Equivalence Between Logic Programs*

Yin Chen[1,2], Fangzhen Lin[3], and Lei Li[2]

[1] Department of Computer Science, South China Normal University, China
[2] Software Institute, Sun Yat-sen University, China
[3] Department of Computer Science, Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong

Abstract. This paper describes a system called **SELP** for studying strong equivalence in answer set logic programming. The basic function of the system is to check if two given ground disjunctive logic programs are equivalent, and if not, return a counter-example. We have used the system to discover some interesting theorems about strong equivalence [Lin and Chen, 2005]. Here we briefly describe how the system can be used to find out whether a given set of rules is strongly equivalent to another, perhaps simpler set of rules.

1 Introduction

The notion of strongly equivalent logic programs was proposed by [Lifschitz et al., 2001]. It has been found useful for tasks such as program simplification. In this paper, we describe a system called **SELP** that can help us answer questions regarding this notion, from simple ones such as "are P and Q strongly equivalent" to more involved ones such as "is there another, preferably simpler logic program that is strongly equivalent to a given one".

The core of the system is checking whether two disjunctive logic programs are strongly equivalent. This is based on [Lin, 2002], which provides a simple mapping from logic programs to propositional theories that reduces strong equivalence to entailment in classical propositional logic. Thus, the problem of strong equivalence checking can be translated into a satisfiability problem in propositional logic, and solved using SAT solvers like *zChaff*. In addition, when two programs are not strongly equivalent, we may want to find some witnesses (counter-example). It is often hard to find a witness manually, but **SELP** can do this automatically: when two programs P_1 and P_2 are found not to be strongly equivalent, it will return a program P, such that $P_1 \cup P$ and $P_2 \cup P$ have different answer sets.

Our motivation for developing **SELP** was to use it to study properties of the notion of strong equivalence. In [Lin and Chen, 2005], we described some results on classes of strongly equivalent logic programs discovered using the system. In this paper, we shall show how the system can help us answer questions of the

* This work was supported in part by HK RGC CERG HKUST6170/04E.

C. Baral et al. (Eds.): LPNMR 2005, LNAI 3662, pp. 442–446, 2005.
© Springer-Verlag Berlin Heidelberg 2005

following form: Given a set P of rules, is there another set of rules of certain property that is strongly equivalent to P?

In [Janhunen and Oikarinen, 2004], a system call LPEQ was developed that can check if two normal programs are strongly equivalent, and was implemented using the answer set logic programming system *smodels*. Besides being implemented using a different technique, our system can deal with normal as well as disjunctive logic programs. Furthermore, our system can construct a counter-example when two programs are not strongly equivalent.

The remainder of this paper is organized as follows. We introduce some basic notions in logic programming in section 2. In section 3, we describe the core of the system, i.e. how to check if two programs are strongly equivalent. In section 4, we show how to find all programs that are strongly equivalent to a given one. Finally, we conclude the paper in section 5.

2 Logic Programming

Let L be a propositional language, i.e. a set of atoms. In this paper we shall consider logic programs with rules of the following form:

$$h_1; \cdots; h_k \leftarrow p_1, \cdots, p_m, not\, p_{m+1}, \cdots, not\, p_n \tag{1}$$

where h_i's and p_i's are atoms in L. So a logic program here can have default negation (*not*), constraints (when $k = 0$), and disjunctions in the head of its rules. In the following, if r is a rule of the above form, we write Hd_r to denote the set $\{h_1, ..., h_k\}$, Ps_r the set $\{p_1, ..., p_m\}$, and Ng_r the set $\{p_{m+1}, ..., p_n\}$. Thus a rule r can also be written as $Hd_r \leftarrow Ps_r, not\, Ng_r$.

The semantics of program is given by answer sets as defined in [**?**]. Two logic programs P_1 and P_2 in L are said to be *equivalent* if they have the same answer sets, and *strongly equivalent* [Lifschitz et al., 2001] (in the language L) if for any logic program P in L, $P \cup P_1$ and $P \cup P_2$ are equivalent.

3 Checking Strong Equivalence Between Two Logic Programs

Lifschitz, Pearce, and Valverde [Lifschitz et al., 2001] showed that checking for strong equivalence between two logic programs can be done in the logic of here-and-there, a three-valued non-classical logic somewhere between classical logic and intuitionistic logic. Lin [Lin, 2002] provided a mapping from logic programs to propositional theories and showed that two logic programs are strongly equivalent iff their corresponding theories in propositional logic are equivalent. This result will be the basis that we are using in this paper for checking if two logic programs are strongly equivalent, and we repeat it here.

Let P_1 and P_2 be two finite logic programs, and L the set of atoms in them.

Theorem 1. *[Lin, 2002] P_1 and P_2 are strongly equivalent iff in the propositional logic, the following two entailments hold:*

$$\{p \supset p' | p \in L\} \cup \Delta(P_1) \models \Delta(P_2), \tag{2}$$

$$\{p \supset p' | p \in L\} \cup \Delta(P_2) \models \Delta(P_1). \tag{3}$$

where for each $p \in L$, p' is a new atom, and for each program P, $\Delta(P) = \{\Delta(r) \mid r \in P\}$, where for each rule r of the form (1), $\Delta(r)$ is the conjunction of the following two sentences:

$$p_1 \wedge \cdots \wedge p_m \wedge \neg p'_{m+1} \wedge \cdots \wedge \neg p'_n \supset h_1 \vee \cdots \vee h_k, \tag{4}$$

$$p'_1 \wedge \cdots \wedge p'_m \wedge \neg p'_{m+1} \wedge \cdots \wedge \neg p'_n \supset h'_1 \vee \cdots \vee h'_k. \tag{5}$$

Notice that if $m = n = 0$, then the left sides of the implications in (4) and (5) are considered to be $true$, and if $k = 0$, then the right sides of the implications in (4) and (5) are considered to be $false$.

Theorem 1 makes it possible to check the strong equivalence between two logic programs using a SAT solver. For instance, to verify (2), it is sufficient to check that both of the following two formulas are satisfied for all $r \in P_2$:

$$\{p \supset p' \mid p \in L\} \cup \Delta(P_1) \cup \{p \mid p \in Ps_r\} \cup \{\neg p' \mid p \in Ng_r\} \cup \{\neg p \mid p \in Hd_r\},$$

$$\{p \supset p' \mid p \in L\} \cup \Delta(P_1) \cup \{p' \mid p \in Ps_r\} \cup \{\neg p' \mid p \in Ng_r \cup Hd_r\}.$$

We implement this idea by using the SAT solver *zChaff*.

If P_1 and P_2 are not strongly equivalent, then *zChaff* will return an assignment that is a counter-example to either (2) or (3), and from this assignment, we can construct a program P such that $P \cup P_1$ and $P \cup P_2$ are not equivalent, i.e. P is a witness of the fact that P_1 and P_2 are not strongly equivalent. The next theorem shows how to do this.

Theorem 2. *Let P_1 and P_2 be two programs, M a model of $\{p \supset p' | p \in L\} \cup \Delta(P_1)$, and not $\Delta(P_2)$. Let M_L and $M_{L'}$ be the two sets of atoms defined as follows:*

$$M_L = \{p \mid p \in L \text{ and } M \models p\}, \tag{6}$$

$$M_{L'} = \{p \mid p \in L \text{ and } M \models p'\}. \tag{7}$$

Then we have

(1) If $M_{L'}$ is not closed under P_2, then $P_1 \cup P$ and $P_2 \cup P$ is not equivalent, where $P = \{p \leftarrow \ \mid p \in M_{L'}\}$.

(2) If $M_{L'}$ is closed under P_2, then $P_1 \cup P$ and $P_2 \cup P$ is not equivalent, where $P = \{p \leftarrow \ \mid p \in M_L\} \cup \{p \leftarrow q \mid p, q \in M_{L'} \setminus M_L, p \neq q\}$.

While the basic function of our system **SELP** is to check whether two given logic programs are strongly equivalent, and if not provides a witness, we do not envision its use this way. Rather, we consider it a tool to systematically study the notion of strong equivalence. We have used this system to discover theorems about strongly equivalent logic programs in [Lin and Chen, 2005]. Here we describe how our system can be used to find whether there is a simpler set of rules that is strongly equivalent to a given set of rules.

4 Finding Strongly Equivalent Logic Programs

One application of **SELP** is about finding out whether there is another, preferably simpler logic program that is strongly equivalent to a given one. For instance, we have seen that the self-loops (loops of length one) like $p \leftarrow p, q$ are strongly equivalent to $\emptyset$. A natural follow-up question is then: what about loops of length two, like $\{(a \leftarrow b), (b \leftarrow a)\}$? Can they be simplified? Similarly, we know that an odd cycle of length one like $\{a \leftarrow not\, a\}$ is strongly equivalent to a constraint like $\{\leftarrow not\, a\}$. Is this also true for an odd cycle of greater length like $\{(a_1 \leftarrow not\, a_2), (a_2 \leftarrow not\, a_3), (a_3 \leftarrow not\, a_1)\}$?

Given a program P in the language L, an obvious way to look for another program in L that is strongly equivalent to P would be to generate all possible programs in L, and call **SELP** on them one by one. This is clearly infeasible even for a program with only three or four atoms. Fortunately, there is a much better way of doing it. Instead of considering all possible sets of rules, we can first find all possible rules that are redundant in the presence of P, i.e. all rules r such that $P \cup \{r\}$ is strongly equivalent to P, and consider sets of these rules only, as the following theorem says.

Theorem 3. *Let P be a logic program in L, and S the set of rules defined as follows:*

$$S = \{r \mid r \text{ is in } L, \text{ and } P \cup \{r\} \text{ and } P \text{ are strongly equivalent }\}.$$

For any program Q in L, if P and Q are strongly equivalent, then $Q \subseteq S$.

Notice that the set S in the theorem includes "trivial" rules like $p \leftarrow p$. As we mentioned in [Lin and Chen, 2005], we need to consider only rules where each atom occurs at most once, i.e. non-redundant rules.

Corollary 1. *Let P be a logic program in L, and S_P the set of rules defined as follows:*

$$S_P = \{r \mid r \text{ is a non-redundant rule in } L, \text{ and } P \cup \{r\} \text{ and } P \text{ are strongly equivalent }\}.$$

For any program Q in L, if P and Q are strongly equivalent, then $Q' \subseteq S_P$, where Q' is obtained from Q by deletion rules that are strongly equivalent to $\emptyset$, and replace each remaining rule r by $Hd_r \setminus Ng_r \leftarrow Ps_r, not\, Ng_r$.

Using this corollary, our system **SELP** finds all programs that are strongly equivalent to a given logic program in two steps:

- generate all possible non-redundant rule r, and check if P is strongly equivalent to $P \cup \{r\}$, thus computing the set S_P,
- for each subset of S_P, check if it is strongly equivalent to P.

For instance, consider again the following set of rules that is in fact a positive loop with length two: $P_1 = \{(a \leftarrow b), (b \leftarrow a)\}$. Using **SELP**, we can see that S_{P_1} consists of the following rules:

$$a \leftarrow b \qquad b \leftarrow a \qquad \leftarrow a, not\, b \qquad \leftarrow b, not\, a$$

As it turned out, there is no subset of S_{P_1} that is strongly equivalent to P_1 yet does not contain P_1, i.e. P_1 cannot be simplified using strong equivalence. Similarly, the odd cycle with length three $\{(a_1 \leftarrow not\, a_2), (a_2 \leftarrow not\, a_3), (a_3 \leftarrow not\, a_1)\}$ is not strongly equivalent to any constraints.

5 Concluding Remarks and Future Work

We have developed a system called **SELP** for studying strong equivalence between logic programs. It plays a central role in discovering some theorems about strong equivalence as described in [Lin and Chen, 2005]. Here we have shown that it can also be used to check if a given set of rules is strongly equivalent to another, hopefully "simpler" set of rules. There are several directions for future work. One of them is to develop a similar system for studying the notion of uniform equivalence between two logic programs [Eiter and Fink, 2003]. In particular, it is interesting to see if theorems like those for strong equivalence discovered in [Lin and Chen, 2005] can be discovered for uniform equivalence.

References

[Eiter and Fink, 2003] Thomas Eiter and Michael Fink. Uniform equivalence of logic programs under the stable model semantics. In Catuscia Palamidessi, editor, *ICLP*, volume 2916 of *Lecture Notes in Computer Science*, pages 224–238. Springer, 2003.

[Gelfond and Lifschitz, 1991] Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991.

[Janhunen and Oikarinen, 2004] Tomi Janhunen and Emilia Oikarinen. Lpeq and dlpeq - translators for automated equivalence testing of logic programs. In Vladimir Lifschitz and Ilkka Niemelä, editors, *LPNMR*, volume 2923 of *Lecture Notes in Computer Science*, pages 336–340. Springer, 2004.

[Lifschitz *et al.*, 2001] V. Lifschitz, D. Pearce, and A. Valverde. Strongly equivalent logic programs. *ACM Transactions on Computational Logic*, 2(4):526–541, 2001.

[Lin and Chen, 2005] Fangzhen Lin and Yin Chen. Discovering classes of strongly equivalent logic programs. In *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence (IJCAI–05)*, 2005. To appear.

[Lin, 2002] Fangzhen Lin. Reducing strong equivalence of logic programs to entailment in classical propositional logic. In *Proceedings of the Eighth International Conference on Principles of Knowledge Representation and Reasoning (KR2002)*, pages 170–176, 2002.

CMODELS – SAT-Based Disjunctive Answer Set Solver

Yuliya Lierler

Erlangen-Nürnberg Universität
yuliya.lierler@informatik.uni-erlangen.de

1 Introduction

Disjunctive logic programming under the stable model semantics [GL91] is a new methodology called *answer set programming* (ASP) for solving combinatorial search problems. This programming method uses answer set solvers, such as DLV [Lea05], GNT [Jea05], SMODELS [SS05], ASSAT [LZ02], CMODELS [Lie05a]. Systems DLV and GNT are more general as they work with the class of disjunctive logic programs, while other systems cover only normal programs. DLV is uniquely designed to find the answer sets for disjunctive logic programs. On the other hand, GNT first generates possible stable model candidates and then tests the candidate on the minimality using system SMODELS as an inference engine for both tasks. Systems CMODELS and ASSAT use SAT solvers as search engines. They are based on the relationship between the completion semantics [Cla78], loop formulas [LZ02] and answer set semantics for logic programs. Here we present the implementation of a SAT-based algorithm for finding answer sets for disjunctive logic programs within CMODELS. The work is based on the definition of completion for disjunctive programs [LL03] and the generalisation of loop formulas [LZ02] to the case of disjunctive programs [LL03]. We propose the necessary modifications to the SAT based ASSAT algorithm [LZ02] as well as to the generate and test algorithm from [GLM04] in order to adapt them to the case of disjunctive programs. We implement the algorithms in CMODELS and demonstrate the experimental results.

2 Syntax of CMODELS

A *Disjunctive program* (DP) is a set of rules with expressions that have the form

$$A \leftarrow B, F \tag{1}$$

where A is the head of the rule and is a disjunction of atoms or symbol $\bot$, B is a conjunction of atoms, and F is a formula of the following form

$$not\ A_1, \ldots, not\ A_m, not\ not\ A_{m+1}, \ldots, not\ not\ A_n$$

where A_i are atoms. We call such rules *disjunctive*. If a head of a rule does not contain disjunction, we call such a rule *normal*. If the formula F of the rule (1) contains an expression of the form $not\ not\ A_i$ then the rule is *nested*, otherwise the rule is *non-nested*. If all rules of a DP are normal we call the program normal.

C. Baral et al. (Eds.): LPNMR 2005, LNAI 3662, pp. 447–451, 2005.
© Springer-Verlag Berlin Heidelberg 2005

Our implementation – system CMODELS – uses the program LPARSE *--dlp-choice* for grounding disjunctive logic programs. The input of CMODELS may include rules of three types. It allows *(i)* non-nested disjunctive rules, *(ii)* choice rules that have the form

$$\{A_0, ...A_k\} \leftarrow A_{k+1}, \ldots, A_l, not\ A_{l+1}, \ldots, not\ A_m \tag{2}$$

where A_i are atoms, and *(iii)* weight constraints of the form

$$A_0 \leftarrow L[A_1 = w_1, \ldots, A_m = w_m, not\ A_{m+1} = w_{m+1}, \ldots, not\ A_n = w_n] \tag{3}$$

where A_0 is an atom or symbol $\bot$; $A_1, \ldots, A_n$ are atoms; L (lower bound); and $w_1 \ldots w_n$ (weights) are integers.

The concept of an answer set for programs containing rules (2) and (3) was introduced in [NS00]. The original rules given to the front end LPARSE *--dlp-choice* allow lower and upper bounds for choice rules and upper bounds for weight rules. They also allow use of literals (negated atoms) in place of atoms. LPARSE *--dlp-choice* translates all the rules to the forms specified above. In CMODELS, choice rules are translated into normal nested rules, and weight constraints are translated with the help of auxiliary variables into normal non-nested rules.[FL05]

Note that CMODELS is the first answer set programming system that allows use of disjunctive and choice rules in the same program.

3 Details on the Modified Algorithms and the Implementation

The implementation is based on definitions of completion, tightness and loop formula for DP introduced in [LL03]. We also refer the reader to [LL03] for formal definitions of a set of atoms satisfying a program, answer set, reduct, and positive dependency graph of DP. The implementation exploits the relationship between completion semantics, loop formulas and answer set semantics for DP. For class of programs called tight models of completion and answer sets are the same. For nontight programs the difference in semantics is due to the cycles (loops) in the program. Loop formulas serve a role of an extension to completion so that the semantics coincide again. Number of loop formulas is exponential and therefore precomputing all loop formulas at once is not feasible, and iterative approach is explored. The correctness of algorithms encoded in CMODELS follows from two theorems.

Theorem for Tight Programs. [LL03] *For any tight DP Π and any set X of atoms, X is an answer set for Π iff X satisfies program's completion $comp(\Pi)$.*

Theorem 1. *Let Π be a DP, M be a model of its completion $comp(\Pi)$, set of atoms $M' \models \Pi^M$, such that $M' \subset M$. There must be a loop of Π under $M \setminus M'$, s.t. M does not satisfy its loop formula.*

Deciding whether a model of the completion is an answer set of disjunctive program is co-NP-complete. Within this implementation of CMODELS we verify that a model of the completion is indeed an answer set by using the minimality requirement of an answer set. We invoke a SAT solver on formula $\Pi^M \cup M^- \cup \neg M$, where (i) Π^M denotes the reduct of Π under M, s.t. its rules are represented as propositional formulas with

the comma understood as conjunction, and $A \leftarrow B$ as the material implication $B \supset A$; (ii) M^- denotes the conjunction of negation of the atoms in Π that do not belong to M; and (iii) $\neg M$ denotes the negation of the conjunction of atoms in M. If this formula is unsatisfied then M is indeed an answer set of Π otherwise some model $M' \subset M$ is returned. Note that $M' \models \Pi$. We call this procedure *minimality test*. It is similar to the procedure introduced in [JNSY00]. [KLP03] introduced a more sophisticated way of verifying whether a model is an answer set using SAT solvers by exploiting some modularity property of the program, that permits splitting verification step on the whole program into verification on its parts. It is a direction of future work to research the applicability of the approach to the case of nested programs.

CMODELS' algorithm is enhanced to verify the tightness of DP at first. In case when a program is tight it performs a completion procedure on the program and uses a SAT solver for enumerating its answer sets, avoiding invocation of minimality test procedure. This way we allow efficient use of SAT solvers in ASP, by analysing program syntactically and identifying in advance disjunctive program involving lower computational complexity.

For nontight programs we adapt ASSAT algorithm [LZ02] to the case of disjunctive programs based on Theorem 2. The modified algorithm follows — *DP-assat-Proc*:

1. Let T be the Completion of Π — $Comp(\Pi)$
2. Invoke SAT solver *SAT-A* to find a model M of T. If there is no such model then terminate with failure.
3. Invoke the minimality test procedure on program Π, and model M with SAT solver *SAT-B* to find model M'. If there is no such model then exit with an answer set M. If there is a model M' then M is not an answer set of Π.
4. Build the subgraph $G_{M \setminus M'}$ of the positive dependency graph of Π induced by $M \setminus M'$. Look for loop L in $G_{M \setminus M'}$, s.t. $M \not\models F_L$, where F_L is a loop formula of L.
5. Let T be $T \cup F_L$, and go back to step 2.

The implementation also adapts another SAT-based answer set programming generate and test algorithm from [GLM04] to the case of nontight disjunctive programs. State-of-the-art SAT solvers are enhanced by the ability of performing backjumping and learning within standard SAT Davis-Logemann-Loveland (DLL) procedure. Backjumping and learning techniques are due to providing DLL procedure with a certain clause. We retrieve the necessary clause from some loop formula of a program that allows us to enhance SAT solver inner computation. The enhanced generate and test algorithm for DP — *DP-generate-test-enhanced-Proc*:

1. Compute completion of Π — $Comp(\Pi)$
2. Initiate SAT solver *SAT-A* with the completion $Comp(\Pi)$. Invoke DLL to find model M of $Comp(\Pi)$. If there is no such model then terminate with failure.

3, 4. The same as Step 3,4 of *DP-assat-proc*.

5. Calculate a clause Cl implied by F_L such that $M \not\models Cl$.
6. Return control to the *SAT-A* procedure DLL by giving Cl as a clause to backjump and learn. Find the next model M of the completion. If there is no such model then terminate with failure. Go back to step 3.

4 Experimental Analyses

Details on the performance of system CMODELS in case of tight disjunctive programs can be found in [Lie05b]. For experimental analysis of CMODELS' performance on non-tight programs we shall specify the algorithmic differences of SAT solvers' invocations. Algorithm *DP-assat-Proc* is implemented in CMODELS using SAT solver MCHAFF[1] in Step 2. Algorithm *DP-generate-test-enhanced-Proc* is implemented in CMODELS with SAT solver SIMO[2] or ZCHAFF[1] invoked in place of *SAT-A* in the procedure. In case of *DP-generate-test-enhanced-Proc* implementation of Step 6 when control is given back to the SAT solver, SIMO and ZCHAFF behave differently. SIMO continues its work with the same search tree it obtained in previous computations, while ZCHAFF starts building a new search tree. In all cases ZCHAFF is used for minimality test procedure.

instance	sat	dlv.5.02.23	cmodels+mchaff	cmodels+zchaff	cmodels+simo	gnt2
qbf7	SAT	15.67	0.01 (23)	0.01 (16)	0.14 (5)	-
qbf8	SAT	92.45	0.01 (23)	0.01 (5)	0.09 (4)	-
qbf9	SAT	7.50	0.01 (33)	0.01 (12)	0.09 (5)	25.77
qbf1	UNSAT	19.81	0.21(10)	0.01 (16)	0.01 (37)	0.001
qbf2	UNSAT	5.43	-	823.98 (19928)	239.68 (26523)	1466.30
qbf3	UNSAT	5.27	-	1779.28 (28481)	193.69 (21260)	-
qbf4	UNSAT	6.83	memory	10.55 (137)	33.64 (663)	-

Fig. 1. CMODELS using MCHAFF, ZCHAFF, SIMO vs. DLV, and GNT on 2QBF benchmark

The first experiment that we demonstrate is 2QBF benchmark. The problem is Σ_2^p-hard. The encoding and the instances of the problem where obtained at the web-site of the University of Kentucky[3]. Figure 1 presents the results. The experiments were run on Pentium 4, CPU 3.00GHz. The columns 3 through 7 present the running times of the systems in seconds with 30 minutes cutoff time. Number in parentheses specifies how often CMODELS invoked the minimality test procedure during its run. In case of satisfiable instances of the problem we can see the payoff in using CMODELS in place of other disjunctive ASP solvers. The picture changes when unsatisfiable instances of the problem come into play. Implementation of *DP-assat-Proc* reaches time limit twice and in case of one instance reaches the memory limit. Implementation of *DP-generate-test-enhanced-Proc* shows better results but as a rule is slower than DLV running time by two orders of magnitude. If we pay attention to the number of minimality test procedure invocations, the slow performance is not surprising. The number of models of the completion is large in case of unsatisfiable instances *qbf2, qbf3* instances and hence all found models must be verified and denied by the minimality test procedure.

The second experiment that we present is the Strategic Company benchmark. The problem is Σ_2^p-hard. We used the encoding and the instances of the problem provided by the benchmark system for answer set programming – Asparagus[4]. Figure 2 presents

[1] `http://www.princeton.edu/~chaff/`
[2] `http://www.star.dist.unige.it/~sim/simo/`
[3] `http://www.cs.uky.edu/ai/benchmark-suite/`
[4] `http://asparagus.cs.uni-potsdam.de/`

inst-ance	dlv.4 5.23	gnt2	cmodels zchaff	cmod-s mchaff	cmod-s simo	inst-ance	dlv.4 5.23	gnt2	cmodels zchaff	cmod-s mchaff	cmod-s simo
160.1	0.64	1.08	0.33	0.40	0.34	125.45	9.03	41.02	-	-	-
160.3	0.87	1.23	0.34	0.40	0.34	105.38	15.55	79.99	315.41	404.72	580.23
75.37	0.51	6.78	1.20	2.49	1.49	155.0	26.15	16.56	-	-	-
150.2	6.66	41.25	1.52	2.10	5.04	135.11	49.01	8.00	191.89	62.25	577.12
150.26	2.24	5.64	5.99	27.04	14.27	155.3	144.00	188.14	43.11	755.12	215.46

Fig. 2. CMODELS using ZCHAFF, MCHAFF, SIMO vs. DLV, GNT on Strategic Company

running times of systems obtained from Asparagus, machine AMD Athlon 1.4GHz PC with 512MB RAM and cutoff time 15 minutes. All given instances are satisfiable. In case of strategic company benchmark there is no clear winner in the performance, but GNT and DLV are in general faster.

References

[Cla78] Keith Clark. Negation as failure. In Herve Gallaire and Jack Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, New York, 1978.

[FL05] Paolo Ferraris and Vladimir Lifschitz. Weight constraints as nested expressions. *Theory and Practice of Logic Programming*, 5:45–74, 2005.

[GL91] Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991.

[GLM04] Enrico Giunchiglia, Yuliya Lierler, and Marco Maratea. Sat-based answer set programming. In *Proc. AAAI-04*, pages 61–66, 2004.

[Jea05] T. Janhunen and et al. *GnT (Generate'n'Test): A Solver for Disjunctive Logic Programs*. 2005. Available under `http://www.tcs.hut.fi/Software/gnt/`.

[JNSY00] T. Janhunen, I. Niemela, P. Simons, and J.H. You. Unfolding partiality and disjunctions in stable model semantics. In *Proc. KR*, 2000.

[KLP03] C. Koch, N. Leone, and G. Pfeifer. Enhancing disjunctive logic programming systems by sat checkers. *Artificial Intelligence*, 151:177–212, 2003.

[Lea05] N. Leone and et al. *A disjunctive datalog system DLV (2005-02-23)*. 2005. Available under `http://www.dbai.tuwien.ac.at/proj/dlv/`.

[Lie05a] Y. Lierler. *CMODELS – a tool for computing answer set using SAT solvers*. 2005. Available under `http://www.cs.utexas.edu/users/tag/cmodels`.

[Lie05b] Yu. Lierler. Cmodels for tight disjunctive logic programs. In *19th Workshop on (Constraint) Logic Programming W(C)LP*, 2005.

[LL03] Joohyung Lee and Vladimir Lifschitz. Loop formulas for disjunctive logic programs. In *Proc. ICLP-03*, pages 451–465, 2003.

[LZ02] Fangzhen Lin and Yuting Zhao. ASSAT: Computing answer sets of a logic program by SAT solvers. In *Proc. AAAI-02*, 2002.

[NS00] Ilkka Niemelä and Patrik Simons. Extending the Smodels system with cardinality and weight constraints. In *Logic-Based Artificial Intelligence*, pages 491–521. 2000.

[SS05] P. Simons and T. Syrjaenen. *SMODELS and LPARSE – a solver and a grounder for normal logic programs*. 2005. `http://saturn.hut.fi/pub/smodels/`.

Author Index

Alferes, José Júlio 356
Angele, Jürgen 26
Anger, Christian 422
Arieli, Ofer 132, 145

Banti, Federico 356
Beierle, Christoph 374
Brogi, Antonio 356
Bruynooghe, Maurice 132, 145
Buccafurri, Francesco 317

Calimeri, Francesco 105
Caminiti, Gianluca 317
Chen, Yin 442
Chesñevar, Carlos I. 158
Cortés-Calabuig, Alvaro 132, 145
Craven, Robert 198

De Bonis, Valerio 432
Dekhtyar, Alex 330
Dekhtyar, Michael I. 330
Dell'Armi, Tina 432
Denecker, Marc 145, 291
Dusso, Oliver 374

Eiter, Thomas 13, 379, 416, 437
Elkabani, Islam 427
Elkhatib, Omar 399
Engan, Iselin 304

Faber, Wolfgang 40, 240, 379, 437
Ferraris, Paolo 79, 119
Fink, Michael 379, 416
Finzi, Alberto 185

Galizia, Stefania 432
Gebser, Martin 53, 422
Gelfond, Michael 172
Godo, Lluis 158
Gottlob, Georg 379
Granata, Luigi 379
Grasso, Giovanni 432
Greco, Gianluigi 379
Grell, Susanne 394
Gressmann, Jean 227
Guo, Hai-Feng 253

Heymans, Stijn 92
Hitzler, Pascal 356

Ianni, Giovambattista 105, 379

Janhunen, Tomi 227, 405

Kakas, Antonis 211, 389
Kałka, Edyta 379
Kern-Isberner, Gabriele 374
Kifer, Michael 1
Konczak, Kathrin 384, 394

Langholm, Tore 304
Lembo, Domenico 379
Lenzerini, Maurizio 379
Leone, Nicola 379, 432
Li, Lei 442
Lian, Espen H. 304
Lierler, Yuliya 447
Lin, Fangzhen 442
Lin, Guohui 369
Linke, Thomas 422
Lio, Vincenzino 379
Liu, Guohua 266
Liu, Lengning 410
Lukasiewicz, Thomas 185

Marek, Victor W. 66
Mercer, Robert E. 227
Michael, Loizos 211
Miller, Rob 211
Moench, Eddie 26
Morales, A. Ricardo 172

Neumann, André 422
Nowicki, Bartosz 379

Odintsov, Sergei 343
Oikarinen, Emilia 405
Onuczko, Curtis 266
Opperman, Henrik 26

Papatheodorou, Irene 389
Pearce, David 343
Pivkina, Inna 66
Pontelli, Enrico 399, 427

Ricca, Francesco 240, 432
Rosati, Riccardo 379
Ruzzi, Marco 379

Schaub, Torsten 53, 227, 394, 422
Senko, Ján 416
Sergot, Marek 198, 389
Simari, Guillermo R. 158
Son, Tran Cao 172, 399, 427
Staniszkis, Witold 379

Terracina, Giorgio 379
Thiele, Sven 227
Tichy, Richard 227
Traxler, Patrick 437
Truszczyński, Mirosław 66, 410
Tu, Phan Huy 172

Van Nieuwenborgh, Davy 92
Van Nuffelen, Bert 132, 145
Vennekens, Joost 291
Vermeir, Dirk 92
Vogel, Ralf 384

Waaler, Arild 304
Wang, Kewen 279
Wenke, Dirk 26
Wu, Gang 369

You, Jia-Huai 266, 369
Yuan, Li Yan 266

Zhang, Yan 279