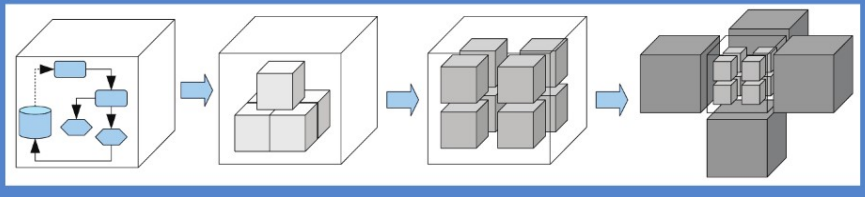
Hot Topics

LNAI 4411

Rafael H. Bordini
Mehdi Dastani
Jürgen Dix
Amal El Fallah Seghrouchni (Eds.)

Programming Multi-Agent Systems

4th International Workshop, ProMAS 2006
Hakodate, Japan, May 2006
Revised and Invited Papers



Springer

Lecture Notes in Artificial Intelligence 4411

Edited by J. G. Carbonell and J. Siekmann

Subseries of Lecture Notes in Computer Science

Rafael H. Bordini Mehdi Dastani
Jürgen Dix Amal El Fallah Seghrouchni (Eds.)

Programming Multi-Agent Systems

4th International Workshop, ProMAS 2006
Hakodate, Japan, May 9, 2006
Revised and Invited Papers

Springer

Volume Editors

Rafael H. Bordini
University of Durham
Department of Computer Science
Durham DH1 3LE, UK
E-mail: R.Bordini@durham.ac.uk

Mehdi Dastani
Utrecht University
P.O.Box 80.089, 3508 TB Utrecht, The Netherlands
E-mail: mehdi@cs.uu.nl

Jürgen Dix
Clausthal University of Technology
Julius-Albert-Str. 4, 38678 Clausthal-Zellerfeld, Germany
E-mail: dix@tu-clausthal.de

Amal El Fallah Seghrouchni
LIP6
104, Avenue du Président Kennedy, 75016 Paris, France
E-mail: amal.elfallah@lip6.fr

Library of Congress Control Number: 2007924245

CR Subject Classification (1998): I.2.11, I.2, C.2.4, D.2, F.3, D.3

LNCS Sublibrary: SL 7 – Artificial Intelligence

ISSN 0302-9743
ISBN-10 3-540-71955-5 Springer Berlin Heidelberg New York
ISBN-13 978-3-540-71955-7 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12048755 06/3142 5 4 3 2 1 0

Preface

These are the proceedings of the 4th International Workshop on Programming Multi-Agent Systems (ProMAS 2006), held on May 9, 2006 in Hakodate (Japan) as an associated event of AAMAS 2006: the main international conference on autonomous agents and multi-agent systems (MAS). ProMAS 2006 was the fourth of a series of workshops that is attracting the increasing attention of researchers and practitioners in multi-agent systems.

The idea of organizing the first workshop of the series was first discussed during the Dagstuhl seminar Programming Multi-Agent Systems Based on Logic (see [6]), where the focus was on *logic-based approaches*. It was felt that the scope should be broadened beyond logic-based approaches, thus giving the current scope and aims of ProMAS [see [4] for the proceedings of the first workshop (ProMAS 2003), [1] for the proceedings of the second workshop (ProMAS 2004), and [3] for the proceedings of the third workshop (ProMAS 2005)]. All four events of the series were held as AAMAS workshops.

Besides the ProMAS Steering Committee (Rafael Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah Seghrouchni), an AgentLink III Technical Forum Group on *Programming Multi-Agent Systems* has been very active in the last couple of years (see http://www.cs.uu.nl/∼mehdi/al3promas.html for details on that group). Moreover, we have edited a book on *Multi-Agent Programming* [2], and ProMAS 2007 will be held with AAMAS 2007 on May 12 or 13, in Honolulu, Hawaii (see http://www.cs.uu.nl/ProMAS/ for up-to-date information about ProMAS).

At the next edition of this workshop series, ProMAS 2007, a Multi-Agent Contest was organized (see [5] and http://cig.in.tu-clausthal.de/AgentContest/). This contest was the third in a series that has been co-organized with the CLIMA workshop series (Computational Logic in Multi-Agent Series). The contest is an attempt to stimulate research in the area of multi-agent programming by *(1) identifying key problems* and *(2) collecting suitable benchmarks* that can serve as milestones for testing agent-oriented programming languages, platforms and tools. A simulation platform has been developed to test MAS which have to solve a cooperative task in a dynamically changing environment.

In addition, preparations for editing a special issue of the *International Journal of Agent-Oriented Software Engineering* are under way. Indeed, we plan to publish this special issue in the Autumn of 2007. It will emphasize the close relation between current research on agent-oriented programming languages and agent-oriented software engineering (*ProMAS meets AOSE*). More information about this special issue can be found on the following Web page: http://www.cs.uu.nl/ProMAS/ProMASSpecialIssue.htm

Finally, we would like to mention the Dagstuhl Seminar on Foundations and Practice of Programming Multi-Agent Systems that recently took place (see

http://www.dur.ac.uk/r.bordini/Dagstuhl06261/). This seminar was organized to bring together researchers interested in programming languages for multi-agent systems, agent-oriented software engineering, and various related aspects such as verification and formal semantics. In this seminar, participants presented their views on the most advanced techniques being currently investigated throughout the world. They also collected the most important open problems of our research community. The seminar was particularly successful in elucidating the relationship between work being done by the *programming languages for multi-agent systems* (ProMAS) research community and the *agent-oriented software engineering* (AOSE) research community. Even though the initiative for this seminar arose from the ProMAS community, many prominent researchers from the AOSE community were attracted as well. This clearly shows the tight connection between ProMAS and AOSE research.

One of the driving motivations behind the ProMAS workshop series (and all associated activities) was the observation that the area of autonomous agents and MAS has grown into a promising technology offering sensible alternatives for the design of distributed, intelligent systems. Several efforts have been made by researchers and practitioners, both in academia and industry, and by several standardization consortia in order to provide new languages, tools, methods, and frameworks so as to establish the necessary standards for a wide use of MAS technology. However, until recently the main focus of the MAS research community has been on the development, sometimes by formal methods but often informally, of concepts (concerning both mental and social attitudes), architectures, coordination techniques, and general approaches to the analysis and specification of MAS. In particular, this contribution has been quite fragmented, without any clear way of "putting it all together," and thus completely inaccessible to practitioners.

We are convinced today that the next step in furthering the achievement of the MAS project is irrevocably associated with the *development of programming languages and tools that can effectively support MAS programming* and the *implementation of key notions in MAS in a unified framework*. The success of MAS development can only be guaranteed if we can bridge the gap from analysis and design to effective implementation. This, in turn, requires the development of fully-fledged and general purpose programming technology so that the concepts and techniques of MAS can be easily and directly implemented.

ProMAS 2006, as indeed ProMAS 2003, ProMAS 2004 and ProMAS 2005, was an invaluable opportunity for leading researchers from both academia and industry to discuss various issues on programming languages and tools for MAS. Showing the increasing importance of the ProMAS aims, the attendance to our workshop has been growing steadily over the years.

This volume of the LNAI series constitutes the official (post-)proceedings of ProMAS 2006. It presents the main contributions that featured in the latest ProMAS event. It contains 12 high-quality accepted and revised papers and, in addition, two invited papers. The structure of this volume is as follows:

Invited papers: We were honored to have Onn Shehory, a leading researcher in the area, giving the invited talk at ProMAS 2006. Onn Shehory has contributed to theoretical research on game-theoretic coalition formation, but also has an interest in agent-oriented software engineering, with significant industrial experience, thus an ideal speaker for the ProMAS workshop series. Subsequently to his talk, he wrote a paper to feature in these proceedings. The paper, based on the invited talk, entitled "A Self-Healing Approach to Designing and Deploying Complex, Distributed and Concurrent Software Systems," discusses a recent project addressing the important area of "self-*" systems. Computing systems with the ability to heal themselves, by diagnosing the cause of failures or inadequate performance and automatically restructuring software components or operating parameters, would make a major impact in the quality of complex dependable systems to be developed in the future.

We also invited Jörg Müller to contribute a paper on his latest work on combining BDI agent and P2P protocols. He wrote the paper, co- authored with Klaus Fischer, Fabian Stäber, and Thomas Friese, entitled "Using Peer-to-Peer Protocols to Enable Implicit Communication in a BDI Agent Architecture." The paper describes research aimed at extending current agent platforms so that agents can interact both through the usual message-based communication but also through document-based communication, as is typical in business processes. Their ideas were practically realized by combining the JACK Agent Platform and the P2P Business Resource Management Framework.

Part I: The first part of these proceedings contains four papers.

The first paper, "Asimovian Multiagents: Applying Laws of Robotics to Teams of Agents and Humans" by Nathan Schurr, Pradeep Varakantham, Emma Bowring, Milind Tambe and Barbara Grosz, investigates the first two laws of Isaac Asimov formulated in the 1940s. It turns out that operationalizing these laws in the context of mixed human-agent teams raises several problems. Among them the uncertainty of agents with respect to their knowledge of the world.

The second paper, "Persistent Architecture for Context-Aware Lightweight Multi-Agent System" by Aqsa Bajwa, Obaid Malik, Sana Farooq, Sana Khalique and Farooq Ahmad, is about an architecture for lightweight devices (e.g., PDAs). An important aspect is to minimize communication latency and to develop a FIPA-compliant context-aware system that can be used for business and e-commerce applications.

The third paper, "Design of Component-Based Agents for Multi-Agent-Based Simulation" by Jean-Pierre Briot, Thomas Meurisse and Frédéric Peschanski, presents a component-based approach to developing agent systems, based on an explicit control flow between components. Complex behaviors can be modelled by composite components. The framework supports both bottom-up as well as top-down approaches.

The fourth paper, "Incorporating Knowledge Updates in 3APL – Preliminary Report" by Vivek Nigam and João Leite, extends the belief base and

the goal base of the 3APL programming language for MAS to use dynamic logic programming, which allows for the representation of changing knowledge. The advantages of the extension include: evolving knowledge and goal bases, the use of strong negation as well as negation as failure, and goals and communication can be more expressive; as a consequence of increased expressiveness, there is also an increase in the complexity of computing an agent reasoning cycle.

Part II: The second part contains four papers.

The first paper, "Comparing Apples with Oranges: Evaluating Twelve Paradigms of Agency" by Linus Luotsinen, Ladislau Bölöni, Joakim Ekblad, Ryan Fitz-Gibbon and Charles Houchin, uses 12 different notions of agency such as rule-based agents, affective agents, and reinforcement agents in order to implement agents that act in an environment similar to artificial life and game environments. The paper provides a comparative analysis of these notions of agency and indicates which type of problems a development team will face if it decides to use a particular agency notion to implement their system.

The second paper, "Augmenting BDI Agents with Deliberative Planning Techniques" by Andrzej Walczak, Lars Braubach, Alexander Pokahr and Winfried Lamersdorf, investigates the application and coupling of state-based planners to BDI agent frameworks. In this proposal, the BDI framework is responsible for plan monitoring and re-planning while the state-based planner is responsible for plan generation.

The third paper, "ALBA : A Generic Library for Programming Mobile Agents with Prolog" by Benjamin Devèze, Caroline Chopinaud and Patrick Taillibert, presents a generic library to be used by mobile agents written in Prolog. This library supports the creation, execution, mobility and communication of agents.

The fourth paper, "Bridging Agent Theory and Object Orientation: Agent-Like Communication Among Objects" by Matteo Baldoni, Guido Boella and Leendert van der Torre, compares the message-sending mechanism between agents and the method-invocation mechanism of objects. They propose an extension of the method-invocation mechanism with ingredients such as the sender identity, the state of interaction protocol, and the roles played in the interactions.

Part III: The third part contains four papers that tackle the issues of validation, debugging and testing agents and MAS. All these papers consider an MAS as a particular type of distributed system in which the active entities, i.e., agents, are autonomous and run concurrently.

The first paper, "Validation of BDI Agents" by Jan Sudeikat, Lars Braubach, Alexander Pokahr, Winfried Lamersdorf and Wolfgang Renz, addresses the issue of testing and debugging BDI-based MAS. The authors examine how the reasoning mechanism inside agent implementations can be checked and how static analysis of agent declarations can be used to visualize and check the overall communication structure in closed MAS. They

also present the corresponding tool support, which relies on the definition of crosscutting concerns in BDI agents and enables both approaches to the Jadex Agent Platform. The second paper, "A Tool Architecture to Verify Properties of Multiagent System at Runtime" by Denis Meron and Bruno Mermet, describes an architecture allowing one to verify the properties of agents and MAS at runtime. The paper defines a notion of *property* and describes the proposed architecture and the way to check MAS.

The third paper, "On the Application of Clustering Techniques to Support Debugging Large-Scale Multi-agent Systems" by Juan A. Botía, Juan M. Hernansáez and Antonio F. Gómez-Skarmeta, situates the problem of debugging distributed MAS by firstly studying the classical approaches for conventional code debugging and also the techniques used in distributed systems in general. From this initial perspective, it tries to situate agent and MAS debugging. It finally proposes the use of conventional data mining techniques like clustering to help, by summarizing, in debugging large-scale MAS.

The fourth paper, "Debugging Agents in Agent Factory" by Rem Collier, describes how debugging has been supported for the Agent Factory Agent Programming Language (AFAPL). This language employs mental notions such as beliefs, goals, commitments, and intentions to facilitate the construction of agent programs that specify the high-level behavior of the agent.

We would like to thank all the authors, the invited speaker, the authors of the second invited paper, the Program Committee members, and reviewers for their outstanding contribution to the success of ProMAS 2006. We are particularly grateful to the AAMAS 2006 organizers for their technical support and for hosting ProMAS 2006.

January 2007

Rafael H. Bordini
Mehdi Dastani
Jürgen Dix
Amal El Fallah Seghrouchni

References

1. Rafael H. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah Seghrouchni, editors. *Programming Multi-Agent Systems: Second International Workshop (ProMAS 2004), held with AAMAS-2004, 20th of July, New York City, NY (Revised Selected and Invited Papers)*, number 3346 in LNAI, Berlin, 2004. Springer-Verlag.
2. Rafael H. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah Seghrouchni, editors. *Multi-Agent Programming: Languages, Platforms and Applications*. Number 15 in Multiagent Systems, Artificial Societies, and Simulated Organizations. Springer-Verlag, 2005.
3. Rafael H. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah Seghrouchni, editors. *Programming Multi-Agent Systems: Third International Workshop (ProMAS 2005), held with AAMAS-2005, 26th of July, Utrecht, the Netherlands (Revised Selected and Invited Paper)*, number 3862 in LNAI, Berlin, 2005. Springer-Verlag.
4. M. Dastani, J. Dix, and A. El Fallah Segrouchni, editors. *Programming Multi Agent Systems (ProMAS 2003)*, number 3067 in LNAI, Berlin, 2004. Springer-Verlag.
5. Mehdi Dastani, Jürgen Dix, and Peter Novak. The Second Contest on Multi-Agent Systems based on Computational Logic. In K. Inoue, K. Satoh, and F. Toni, editors, *Proceedings of CLIMA'06, Revised Selected and Invited Papers*, number 4371 in LNAI, pages 156–187, Hakodate, Japan, 2006. Springer-Verlag.
6. Jürgen Dix, Michael Fisher, and Yingqian Zhang. Programming Multi Agent Systems based on Logic. Technical Report Dagstuhl Seminar Report 361, IBFI GmbH, Schloß Dagstuhl, 2002.

Organization

Organizing Committee

Rafael H. Bordini (University of Durham, UK)
Mehdi Dastani (Utrecht University, The Netherlands)
Jürgen Dix (Clausthal University of Technology, Germany)
Amal El Fallah Seghrouchni (University of Paris VI, France)

Program Committee

Suzanne Barber (University of Texas at Austin, USA)
Lars Braubach (University of Hamburg, Germany)
Jean-Pierre Briot (University of Paris 6, France)
Keith Clark (Imperial College, UK)
Rem Collier (University College Dublin, Ireland)
Yves Demazeau (Institut IMAG - Grenoble, France)
Frank Dignum (Utrecht University, The Netherlands)
Michael Fisher (University of Liverpool, UK)
Jorge Gómez-Sanz (Universidad Complutense Madrid, Spain)
Vladimir Gorodetsky (Russian Academy of Sciences, Russia)
Benjamin Hirsch (TU Berlin, Germany)
Shinichi Honiden (NII, Tokyo, Japan)
Jomi Hübner (Universidade Regional de Blumenau, Brazil)
João Leite (Universidade Nova de Lisboa, Portugal)
Jiming Liu (Hong Kong Baptist University, Hong Kong)
John-Jules Meyer (Utrecht University, The Netherlands)
Oliver Obst (Koblenz-Landau University, Germany)
Gregory O'Hare (University College Dublin, Ireland)
Andrea Omicini (University of Bologna, Italy)
Agostino Poggi (Università degli Studi di Parma, Italy)
Alexander Pokahr (University of Hamburg, Germany)
Chris Reed (Calico Jack Ltd., UK)
Birna van Riemsdijk (Utrecht University, The Netherlands)
Sebastian Sardina (RMIT University, Australia)
Ichiro Satoh (NII, Kyoto, Japan)
Onn Shehory (IBM Haifa Research Labs, Haifa University, Israel)
Kostas Stathis (City University London, UK)
Simon Thompson (BT, UK)
Leon van der Torre (CWI, The Netherlands)
Paolo Torroni (University of Bologna, Italy)
Gerhard Weiss (Technische Universität München, Germany)

Michael Winikoff (RMIT University, Melbourne, Australia)
Cees Witteveen (Delft University, The Netherlands)

Additional Reviewers

Peter Novak (Clausthal University of Technology, Germany)
Damien Pellier (Université Paris 5, France)
Yingqian Zhang (Clausthal University of Technology, Germany)

Table of Contents

Invited Papers

Part I

Part II

Part III

Invited Papers

A Self-healing Approach to Designing and Deploying Complex, Distributed and Concurrent Software Systems*

Onn Shehory

IBM Haifa Research Lab, Mount Carmel
31905 Haifa, Israel
Onn@il.ibm.com

Abstract. Software systems have become too complex to manage and fix manually. An emerging approach to overcome this problem is software self-healing. Studies in related disciplines have offered several self-healing solutions, however these usually address a single class of problems, or they are not applicable in fielded systems. To address the industrial need for software self-healing, we have initiated the EU SHADOWS project. This project concentrates on self-healing of complex systems, extending the state-of-art in several ways. It introduces pioneering technologies to enable the systematic self-healing of classes of failures which are not solved by other approaches. It additionally introduces an approach to the integration of several self-healing technologies in a common solution framework. It also adopts a model-based approach, where models of desired software behavior direct the self-healing process. Combined, these allow for lifecycle support of software self-healing, applicable to both academic and industrial systems.

Keywords: Complex software systems, Self-healing, Autonomic computing.

1 Introduction

Our society has become dependent on the correct functioning of complex software systems. Classical software assurance methods are limited in their ability to address the increasing scale—in both size and complexity—of contemporary computer systems. This negatively affects system quality and total cost of system ownership. Recent research on self-management and autonomic computing introduces new possibilities for developing highly-reliable self-managed complex software systems, while reducing cost-of-ownership. Computational systems able to manage themselves are commonly called *self-** systems. Current research in this area is still in its early stages. We believe that independent solutions for self-* systems cannot be applied straightforwardly in industrially relevant systems, and that industrially applicable solutions must integrate several technologies in a common framework.

Self-healing refers to the capability of a software system to automatically diagnose and heal the root-cause of its failures and performance problems, and prevent them

* This research is funded in part by the European Commission via the SHADOWS project under contract No. 035157.

R.H. Bordini et al. (Eds.): ProMAS 2006, LNAI 4411, pp. 3–13, 2007.

from reappearing. This is performed through structural modifications of the software system and its operating parameters. We have initiated an EU project (SHADOWS [1]) that concentrates on the self-healing of complex systems, extending the state-of-art in several ways. Firstly, it introduces pioneering technologies to enable systematic self-healing of classes of failures which are not solved by other approaches. Additionally, it integrates several technologies which address different classes of problems and work at different abstraction levels, to provide a common solution framework. Further, it implements a model-based approach, where models of desired software behavior govern the self-healing process throughout the system design phase and the system deployment phase.

The SHADOWS project consists of nine partners. Some partners are research organizations, whereas others are industrial software developers. This combination allows the research partners not only work together to create a unified self-healing solution, but also to validate the solution in the field, in several different application domains, thus delivering a higher-quality solution.

At the time of writing this document, the project has been running for only a few months. Therefore, this article will introduce the background, the problem, the underlying technologies, and the approach and directions taken, however concrete results are not available yet. Intermediate results of the project will be posted at its web site [1].

2 Problem Statement

Classical software assurance introduces two main classes of techniques for increasing system reliability [2][3]. The first class includes rigorous development and testing methodologies; these increase the quality of single components and of their integration [4][5]. The second class includes fault tolerance mechanisms [6][7], which can guarantee the reliability of the system in the presence of faults. The classical methods have proven very useful in the past, and are still widely used. However, they do not scale well to address the increase in size and complexity of contemporary computer systems. This negatively affects the total cost of system ownership. To address the limited capability of classical techniques, self-management and autonomic computing [8][9] methods are being studied. These new research directions introduce new possibilities for developing highly-reliable self-managed complex software systems, while reducing cost-of-ownership.

Self-management of computational systems has already been envisioned in the past, notably by Turing and von-Neumann [10][11]. Only recently, as a result of market needs, attention to self-management has returned. Self-managed systems are commonly denoted as *self-** systems; the asterisk may indicate a variety of attributes, e.g., self-awareness, self-configuration, self-diagnosis etc. Current research in this area has explored multiple directions, approaches and problems, however it is still in its early stages and has focused mostly on academic settings. Among the approaches to self-management one finds reconfigurable architectures, consistency management, code relocation, control theory, agent-based systems, game theory, and others [12][13][14][15][16][17][18][19][20]. Below we briefly survey the primary research directions in the field of self-healing.

3 Related Work

Reconfigurable architectures are systems capable of dynamically modifying their structure. This is done using strategies that are automatically applied when a failure occurs [21][22][23]. Examples of such capabilities are the dynamic addition of redundant servers, the ability to modify the allowed bandwidth for a connection, the addition and removal of components, and the dynamic modification of the connections between components. These self-healing techniques are based on architectural reconfiguration focused on non-functional system properties, such as service availability and performance [24][25][26].

Consistency management techniques are runtime techniques for maintaining the system in a consistent and legal state. Examples include rollback and resume techniques [27] and failure masking techniques. A solution based on failure masking and reconfigurable architectures is presented in [28]. That work introduces a self-healing system that can discover, and connects to, new service providers while devices move through different environments.

Code modification and migration techniques dynamically modify an application's code to repair diagnosed faults. Work in [29] demonstrates the use of code mobility techniques to dynamically move code to different locations, in order to repair faults. A preliminary study in this theme suggests the replacement of the implementation of an active system component with a new implementation, while maintaining the availability of the component's functionality [30].

In our research we study several extensions to the state-of-the-art. We develop techniques that enable the self-healing of problem areas for which no other approach provides an adequate solution. Specifically, we target the following:

- Self-healing of concurrency problems, for which no solution is available currently.
- Self-healing of functional problems is addressed by multiple studies. In our project we develop an innovative automated model-based approach: the appropriate healing strategy is selected based on comparison of fault models to information monitored at healthy-system runtime.
- Self-healing of performance problems was previously addressed, to some extent, by reconfigurable architectures. We incorporate sophisticated statistical and machine learning methods into the self-healing strategy. This allows healing not only the system, but the healing strategy itself, to increase the effectiveness of the self-healing.

In addition to the specific problem domains and approaches we address, our project integrates several technologies. It also introduces a shared model-based approach that works at different levels of abstraction and addresses several parts of the system lifecycle. Furthermore, we target industrial applications, moving the self-healing paradigm from academia to the field.

4 Solution Approach

Independent self-* solutions studied to date are typically not applicable straightforwardly to industrial systems. Therefore, our approach does not limit itself to studying new solutions. Rather, we integrate several solutions in a common framework. Our goal is to allow self-healing by means of automatic diagnosis and healing of the root-cause of system failures and performance problems. We achieve this through structural modifications of the software system and its operating parameters. To provide solutions to the self-healing problem, we target three underlying requirements for software quality and reliability, as follows:

(1) Adherence to functional requirements;
(2) Robust performance;
(3) Safe utilization of concurrency.

Generically, self-healing is a repetitive four-stage process: problem detection, root-cause analysis, healing and assurance. The detection stage reveals the presence of a problem. Root-cause analysis identifies the fault that caused the problem. The healing stage provides problem remediation. Lastly, the assurance stage examines the healing to ensure that it solves the problem without introducing new problems.

Our self-healing solution follows the four-stage approach, however it additionally combines several self-healing technologies: concurrency testing, fault prediction, automatic setting of performance thresholds, and capturing and healing of faults. While each of these technologies can be applied in isolation, their integration and application across the software lifecycle will provide a comprehensive solution for developing self-healing systems.

4.1 Technical Foundations

The SHADOWS project will produce a well-integrated set of technologies, embodied in tools that will support a methodology for developing systems capable of self-healing. Our solution will enable self-healing throughout the software lifecycle so that developers and system administrators can benefit from the results of the project across multiple dimensions of use. These dimensions include:

- Self-healing of software both at design time and at testing and deployment time.
- Self-healing at the component level, as well as at the system level.
- Self-healing for multiple types of errors (performance, function, concurrency).

The self-healing capabilities we develop are based on novel algorithms for the detection, prediction and classification of faults, and for performing a respective corrective action. The detection and prediction activities are performed against models of desirable behavior. Figure 2 presents a conceptual view of our self-healing framework. The framework includes a model repository, a monitoring layer, an analysis layer, and a correction layer. The model repository includes models for safe concurrency, functional requirements, and performance behaviors. The detection and prediction of the problems, manifested by deviations from the models, is performed by the monitoring layer A. The classification and identification of the problems is

performed by the analysis layer B. Detection and analysis of a problem by layer B leads to the invocation of corresponding corrective action (either by layer C at design time, or by layer D at deployment time). Such corrective actions may then feed into the managed system or influence its working environment.

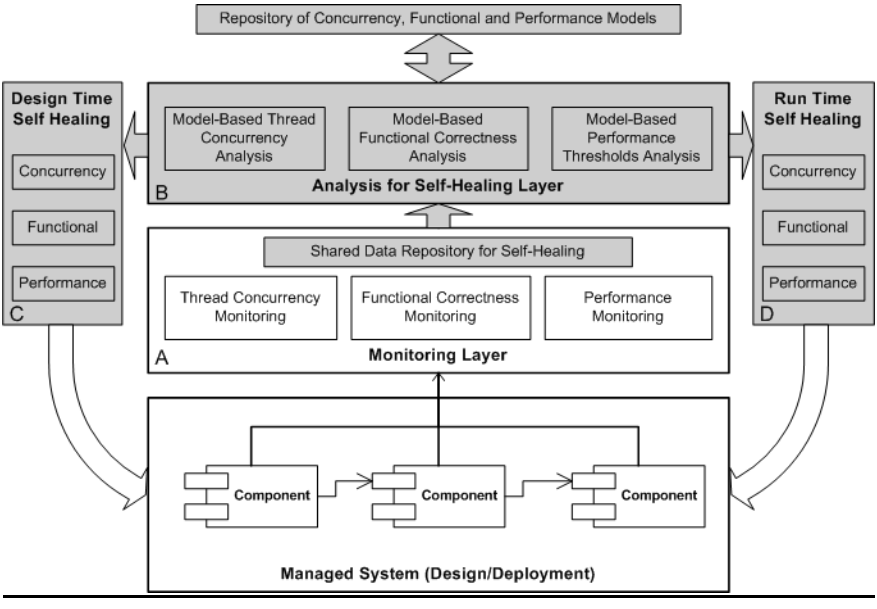


Fig. 1. The architecture of the integrated self-healing framework. The grey elements are our new technologies. The white elements are pre-existing technologies.

Following, we describe our self-healing framework and techniques in more detail.

4.2 Self-healing of Concurrency Problems

A model for safe concurrency was recently introduced in [31][32]. That model covers a comprehensive set of unsafe code patterns that may lead to race conditions, deadlocks, and other concurrency problems. The model is based on an abstraction of unsafe scheduling and synchronization of threads. It includes both generic unsafe patterns and application-specific patterns. In our current study we further extend the safe concurrency model by identifying new domain-specific unsafe patterns.

To identify and predict concurrency-related problems, we develop algorithms that examine program code at coding and testing time, compare it to the safe concurrency models, and detect unsafe concurrency patterns in the code. We employ static identification algorithms that analyze the code for structure and internal dependencies. We further employ dynamic identification algorithms that detect runtime concurrency-related events (e.g., access to global variables). By monitoring such events we can determine whether the program has incurred, or is about to incur, an unsafe behavior. The dynamic identification algorithms use various heuristics to increase the probability of exposing concurrency-related problems.

Once unsafe concurrency problems are detected, we need to resolve them. Our approach to resolving such problems is to automatically introduce scheduling and synchronization statements to affect the concurrency behaviors of the code. This healing process may be applied during the design, testing, and deployment stages of the system lifecycle. For example, if an access to a common resource requires the acquisition of a lock that is not present in the code, a lock statement should be added to fix the problem. Our approach further requires that, upon adding new statements to the code, the result should be verified to make sure that the fixes have the sought effect.

4.3 Self-healing of Functional Problems

Our approach to self-healing of functional problems is based on three types of models: system, fault, and component models. The system model specifies behavioral properties that must be satisfied by the system. These properties can be provided by the user, however can also be extracted from the code and its specifications, when these are available. The fault model specifies the types of faults that can be identified and repaired by our functional self-healing solution. We specifically address fault types related to incompatibility of components which were designed and implemented separately but need to work together as part of an integrated system. The component models describe the behavior of the system's components. They include invariants that should hold during normal system operation; they also include finite state machines describing normal operation inter-component interaction patterns. Component models are provided by the Behavior Capture and Test (BCT) technology [33], which generates and updates them based on behavior observations.

At normal system operation, the BCT technology generates and updates the component models. Upon violation of the system model, the information collected by BCT during the faulty time is checked against the component models. Invariants violations indicate which components misbehaved, and execution traces show the interactions corresponding to the failure. Combined with the component models, this information is checked against the fault model to identify fault models of relevance. Once a specific fault class is identified, appropriate self-healing can be activated.

Our approach to self-healing of functional problems is as follows. Upon fault detection at the system level, healing is applied at the intra- and inter-component levels. This is performed by enabling components with self-healing capabilities. Such enablement is done using reconfiguration algorithms which are implemented either within the component, or externally. In some cases, algorithms for healing specific components are not available, however reconfiguration at the systems level may still be applied to resolve the problem.

4.4 Self-healing of Performance Problems

Our approach to performance self-healing is based on two models: system-level performance model and component-level performance models. Correlations among these can serve as indicators to dissatisfactory performance behavior. The system-level performance model specifies system performance requirements in terms of compliance with performance goals. Commonly, this specification is provided via a

Service Level Objective (SLO). SLOs specify conditions on, and expectations from, system-level operational parameters. Component-level performance models specify requirements on component-level operational parameters, commonly expressed by thresholds set on these parameters. For example, in software that manages a storage server, an SLO may specify that the number of end-to-end transactions per second must be at least 500. A related component-level threshold may specify a lower bound on the number of disk I/Os per second.

In diagnosing performance problems we rely on correlating system-level SLO violations with the underlying component-level threshold violations. Yet, the general case of this problem is very challenging and usually does not have an optimal solution. This means that performance diagnostics inherently include false alarms. False positive alarms occur when a threshold is violated at the component level, but the violation does not result in a violation of the SLO (and thus the system behaves normally). False negative alarms occur when an SLO is violated (and thus the system does not behave in an acceptable manner) but no component-level violation is detected. Both types of false alarms are undesirable as they provide false indications of performance problems. We address this undesirable condition by developing algorithms that, in addition to correlating the system-level model with the component-level model, allow reduction in the level of false alarms [34]. Our false alarms model accounts for both false negative and false positive alarms.

Upon diagnosing a performance problem (an SLO violation) our self-healing algorithms compute and set improved component-level parameters to prevent future violations. The calculation of the new component-level parameters is performed in two stages. Initially, in cases where the number of component-level parameters is large, feature selection techniques are employed to converge on a smaller set of relevant parameters. Then, time series of SLO violations, threshold violations, and operational parameters are used as input to correlation algorithms for computing the optimized parameters. This type of performance optimization can be performed both at design time, to arrive at an optimized system configuration, and at run time, to allow dynamic automated self-healing.

4.5 Integration of the Self-healing Solutions

Although concurrency, functional and performance problems (and solutions) are fundamentally different, they do interfere and interact. Therefore, the self-healing technologies we provide are to be integrated at four main levels: the logical level (models and problem analysis techniques), the technological level, the design level (tools and development environments) and the methodological level.

The concurrency, functional and performance models for problem analysis rely on different information. However, they have strong complementarities and synergies, and we exploit these in our solution. A system under development can be automatically healed to correct potential concurrency and performance problems. The respective healing techniques might interfere with each other. For instance, changing the code to avoid concurrency problems can lead to performance problems. Therefore it is necessary that healing techniques be coordinated to reduce interference. We facilitate coordination by providing each healing technique with access to the models of the others. By this, we allow early identification of the consequences of fixes

suggested by the healing techniques. In some cases, it is possible to improve the results of healing techniques by letting them to interact and collaborate. For instance, a performance problem can be caused by an inefficient implementation of concurrency and can be healed by modifying the concurrency structure, thus improving performance without affecting concurrency.

The healing of concurrency, functional and performance problems shares several underlying technologies. In particular**,** all require monitoring infrastructure and actuating technologies. The monitoring infrastructure is used to collect run-time data over the temporal and functional behavior of the monitored system. The collected data are then analyzed to detect problems and find their root-causes. Run-time actuation is used for activating healing procedures. In our self-healing approach, components implement healing extensions; these allow modification of both their internal and interaction behavior. The healing of both performances and functional problems will use these actuation technologies to apply changes.

The technologies we develop address different problems, but will share the same methodological approach to the design and implementation of a self-healing system. The integrated methodology will focus on (1) coding style for facilitating healing at development time, (2) methods for writing specification documents that can be easily checked by inspection engines, (3) methods for implementing features that support run-time healing, (4) guidelines for implementing components that facilitate monitoring of their behavior. In brief, the methodology will provide a comprehensive set of guidelines to follow along the entire self-healing software lifecycle.

4.6 Evaluation Methodology

To evaluate the self-healing technologies we develop, we will be using industrial case studies. The new technologies will be evaluated using sounding boards provided by the industrial partners. The target applications for the evaluation are: (embedded) consumer electronics software systems, telecommunications software systems, air traffic control software systems, and mobile and web-based software systems. To evaluate the technologies in a systematic and generic manner, we are developing an evaluation methodology. Using such a methodology, we expect our solutions to yield a positive increase in system reliability, measured by (1) a reduction in the number of functional errors (goal: 20%), (2) a reduction in the average number of performance problems (goal: 15%) and in network traffic related to performance problem detection (goal: 20%), (3) a reduction in the number of and concurrency-related errors (goal: 20%). These measures are based on early experiments performed by our partners. The relative improvement is measured in comparison with systems in which our self-healing solutions are not implemented.

5 Summary

Our self-healing research within the SHADOWS project introduces a new approach to the development of robust complex software systems, building on technologies brought in by and experience of our partners. We develop standalone technologies and integrate them into a coherent framework and provide a design methodology to be

used in conjunction with the framework. These should allow self-healing to become accessible to software developers, system administrators, and users of IT systems at large. To achieve our goals, we perform the following:

- Develop self-healing technologies for concurrency, functional and performance problems. These include concurrency testing, fault prediction, automatic setting of performance thresholds, and capturing and healing of faults. This is performed based on the background technologies contributed by the partners.
- Establish a new self-healing system design paradigm as well as a methodology to guide in the use of the new paradigm.
- Implement efficient CASE tools for designing and managing self-healing systems, to be used in conjunction with the self-healing methodology.
- Test the methodology and the tools in several software environments in the field.
- Develop an evaluation methodology and criteria for self-healing system design, and evaluate and quantify the results of the project using these criteria.
- Use internal feedback from partners to refine the methodology and the tools.

These research and development activities should not only provide a self-healing techniques, methodology and tools, but also support their adoption in industrial applications. We believe that the SHADOWS project will provide initial steps towards this direction, however further research and standardization afforest will be required to make self-healing software part of mainstream software engineering.

References

[1] https://sysrun.haifa.il.ibm.com/shadows/index.html
[2] *Software Reliability: Measurement, Prediction, Application*, J.D. Musa, J.D., A. Iannino and K. Okumoto, McGraw–Hill, Professional Edition: Software Engineering Series, 1990.
[3] *Metrics and Models in Software Quality Engineering (2nd Edition),* Stephen H. Kan, Addison-Wessley, 2002.
[4] *Software Quality Assurance: From Theory to Implementation*, D. Galin, Pearson Education, 2003.
[5] *The Art of Software Testing (2nd Edition),* G. Myers, Wiley, 2004.
[6] *Distributed Systems: Principles and Paradigms,* A.S. Tanenbaum and M. van Stee, Prentice-Hall, 2002.
[7] *Fault Tolerant System Design,* S. Levi and Ashok Agrawala, McGraw-Hill, New York, 1994.
[8] *The Dawn of Autonomic Computing,* A.G. Ganek and T.A. Corbi, The Dawn of the Autonomic Computing Era, IBM Systems Journal, Vol. 42(1), 2003.
[9] *The Berkeley/Stanford Recovery-Oriented Computing Project*, http://roc.cs.berkeley.edu/
[10] *Theory of self-Reproducing Automata*, J. von-Newmann, A.W. Burks, Ed., University of Illinois Press, 1966.
[11] *Intelligent Machinery*, A.M. Turing, 1948, Report for National Physical Laboratory, in *Machine Intelligence 7*, Eds. B. Meltzer and D. Michie (1969).

[12] *Complexity and Emergent Behaviour in ICT Systems,* S. Bullock and D. Cliff,, HP Labs report HPL-2004-187, 2004, http://www.hpl.hp.com/techreports/2004/HPL-2004-187.html.

[13] *Towards a Paradigm Change in Computer Science and Software Engineering: a Synthesis,* F. Zambonelli and V. Parunak, The Knowledge Engineering Review, Vol. 18:4, 329–342. 2004.

[14] Proceedings of SELF-STAR: International Workshop on Self-* Properties in Complex Information Systems, 31 May - 2 June 2004, University of Bologna, http://www.cs.unibo.it/self-star/program.html

[15] *Recovery Oriented Computing: Motivation, Definition, Techniques, and Case Studies*, D. Patterson et al, Computer Science Technical Report UCB//CSD-02-1175, U.C. Berkeley, March 15, 2002 http://roc.cs.berkeley.edu/papers/ROC_TR02-1175.pdf.

[16] *Self-Management: The Solution to Complexity or Just Another Problem?*, K. Herrmann, G. Mühl, and K. Geihs, IEEE Distributed Systems Online, V 6(1)1, 2005.

[17] *Efficient and Transparent Instrumentation of Application Components Using an Aspect-oriented Approach*, M. Debusmann and K. Geihs, Proc. 14th IFIP/IEEE Workshop on Distributed Systems: Operations and Management (DSOM 03), LNCS 2867, Springer-Verlag, 2003, pp. 209-220.

[18] *Research Perspectives in Self-Healing Systems,* D. Tosi, Technical Report LTA:2004:06, University of Milano at Bicocca, July 2004, http://www.lta.disco.unimib.it/doc/ei/pdf/lta.2004.06.pdf

[19] *Open Issues in Self-Inspection and Self-Decision Mechanisms for Supporting Complex and Heterogeneous Information Systems*, M. Colajanni, M. Andreolini and R. Lancellotti, Proceedings of. Int'l SELF-STAR 2004 [13], http://www.cs.unibo.it/self-star/papers/colajanni.pdf

[20] *Dynamic Configuration of Resoubrce-Aware Services*, V. Poladian, J. Pedro Sousa, D. Garlan, and M. Shaw, Proceedings of the 26th International Conference on Software Engineering (ICSE), Edinburgh, Scotland, May 2004.

[21] *Self-organising software architectures for distributed software systems*, I. Georgiadis, J. Magee, and J. Kramer. In proceedings of the first workshop on Self-healing systems, pages 33-38. ACM Press, 2002.

[22] *Architecture style requirements for self-healing systems*, M. Mikic-Rakic, N. Mehta, and N. Medvidovic. In proceedings of the first workshop on self-healing systems, pp. 49-54, ACM Press, 2002.

[23] *An architecture-based approach to self-adaptive software*, P. Oreizy, M. Gorlick, R. Taylor, D. Heimhigner, G. Johnson, N- Medvidovic, A. Quilici, D. Rosemblum, and A. Wolf. IEEE Intelligent Systems, 14(3):54-62, May/Jun 1999.

[24] *An infrastructure for multiprocessor run-time adaptation*, J. Appavoo, K. Hui, M. Stumm, R. Wisniewski, D. Da Silva, O. Krieger, and C. Soules. In proceedings of the first workshop on self healing systems, pages 3-8, ACM Press, 2002.

[25] *Model-based adaptation for self-healing systems*, D. Garlan, B. Schmerl. In proceedings of the first workshop on self-healing systems, pages 27-32, ACM Press, 2002.

[26] *Using process technology to* control *and coordinate software adaptation,* G. Valetto and G. Kaiser. Proc 25th international conference on software engineering, pp. 262-272, IEEE Computer Society, 2003.

[27] *Principles of transaction-oriented database recovery*, T. Haerder and A. Reuter. ACM Computer Surveys, 15(4):287-317, 1983.

[28] *Understanding self-healing in service-discovery systems*, C. Dabrowski and K. Mills. In proceedings of the first workshop on self-healing systems, pages 15-20, ACM Press, 2002.

[29] *Understanding code mobility,* A. Fuggetta, G. Picco and G. Vigna. IEEE Transactions on Software Engineering, 24(5):342-361, 1998.

[30] *Containment units: a hierarchically composable architecture for adaptive systems*, J. Cobleigh, L. Osterweil, A. Wise, and B.S. Lerner. Proc. of 10th ACM SIGSOFT symposium on Foundations of Software Engineering, pages 159-165, ACM Press, 2002.

[31] *Multithreaded Java program test generation*, E. Farchi, Y. Nir, G. Ratsaby and S. Ur: IBM Systems Journal 41(1): 111-125 (2002)

[32] *Concurrent Bug Patterns and How to Test Them*, E. Farchi, Y. Nir and S. Ur., Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'03), p. 286b

[33] *Behavior Capture and Test: Automated Analysis of Component Integration,* L. Mariani and M. Pezzè, Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems, IEEE Computer Society, Shangai (China), 16-20 June, 2005.

[34] *Automated and Adaptive Threshold Setting: Enabling Technology for Autonomy and Self Management*, D. Breitgand, E. Henis and O. Shehory, Proc. 2nd Intl. Conference on Autonomic Computing (ICAC'05), Seattle, June 2005.

Using Peer-to-Peer Protocols to Enable Implicit Communication in a BDI Agent Architecture

Klaus Fischer[1], Jörg P. Müller[2,3], Fabian Stäber[3], and Thomas Friese[3]

[1] DFKI GmbH
Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany
klaus.fischer@dfki.de
[2] Dept. of Computer Science
Clausthal University of Technology
Julius-Albert-Str. 4, D-38678 Clausthal-Zellerfeld, Germany
joerg.mueller@tu-clausthal.de
[3] Siemens AG, Corporate Technology, Information and Communications
Otto-Hahn-Ring 6, D-81739 München, Germany
{fabian.staeber.ext,th.friese}@siemens.com

Abstract. The objective of the research described in this paper is to extend current agent platforms in order to provide both explicit, message-based and implicit, document-based communication and co-ordination within a uniform framework, and to make this unified framework available for the agent-oriented design and enactment of business processes. This is achieved by interfacing a BDI agent platform with an underlying peer-to-peer (P2P) platform, where the P2P framework is used to virtualize certain sections of the belief sets of the BDI agents; after a review of existing approaches to integrate multiagent with P2P concepts, a prototype technical realization is presented using two state-of-the art platforms: the Jack BDI agent platform and the P2P Business Resource Management Framework (BRMF) platform.

1 Introduction

With multiagent systems and technologies becoming more mature, they are finding their way into business applications, such as business process management (BPM) or collaborative product development (see [1], [2]). What makes multiagent system appealing both from a modeling and a runtime angle is their ability to provide natural mappings to concepts usually found in BPM models, such as organizations, roles, and goals, and their intrinsic support to flexible business service composition and loosely coupled coordination and cooperation, as they are often found in collaborative, cross-enterprise business environments.

However, there are some fundamental limitations in the level of support current agent platforms can provide for the execution of business processes. In particular, popular platforms such as Jade[1] or Jack Intelligent Agents™ [2] base their

[1] http://jade.tilab.com
[2] http://www.agentsoftware.com.au

R.H. Bordini et al. (Eds.): ProMAS 2006, LNAI 4411, pp. 15–37, 2007.

interaction models on the paradigm of message-based communication between agents. This is well suitable for business processes that are clearly structured according to e.g., FIPA[3] interaction protocols [3]. Example for processes that can be easily supported by today's agent-based solutions are for example order management or procurement processes, which are complex in that there is a wealth of situation-dependent choices regarding the behavior of the individual participants, but which are structured in that interaction follows clear rules and protocols. However, today's agent platforms and the traditional speech-act- and protocol-based messaging are less suitable to model and support processes that are less structured, and more event-based.

It is our claim that agent-based *explicit* messaging needs to be complemented by an *implicit*, blackboard-style communication and coordination paradigm to be able to support unstructured, document-centric and event-driven business processes. In the collaborative product design use case, a suitable paradigm for the interactions within the supplier network is that of a peer-to-peer (P2P) organization where each party maintains copies of the documents or models in question, and is notified when another party changes one of the documents. The negotiation comes to an end when each party agrees to the current set of documents available. The idea to describe a process by reactions to changes of documents is radically different from traditional business process design.

The objective of the research described in this paper is to extend current agent platforms in order to provide both explicit, message-based and implicit, document-based communication and co-ordination within a uniform framework, and to make this unified framework available for the agent-oriented design and enactment of business processes. In particular, we found a suitable paradigm for efficient implementation of implicit, decentral coordination and resource management within the P2P community.

In this paper we describe a first step towards combining the use of BDI agents to model and enact CBPs process flows with the use of a P2P platform to enable event- and document-driven "publish-subscribe style" collaboration. In doing so, we can provide architects of business application with the concepts and tools to integrate both process-centric and event-/document-centric business collaboration requirements within one unifying architecture. Technically, the first step of this endeavor is achieved by interfacing a BDI agent platform with an underlying P2P platform, where the P2P framework is used to virtualize certain sections of the belief sets of the BDI agents; a technical realization concept is presented using two state-of-the art platforms: the Jack BDI agent platform and the P2P Business Resource Management Framework (BRMF) platform.

The paper is structured as follows: In Section 2, we introduce the Collaborative Product Development (CPD) application used throughout this paper to describe the concepts. Section 3 outlines related work in using multiagent system and P2P concepts for business applications. In Section 4, existing approaches to combine agent and P2P computing are discussed, and the main

[3] http://www.fipa.org

aspects of an integration of the two concepts are presented. Section 5 describes the overall concept of belief base virtualization; a sketch of the implementation of the integration is provided in Section 6. Section 7 illustrates how our concepts can be beneficially used in the collaborative product development application from Section 2. The paper ends with a discussion of results and future research opportunities.

2 Application Scenario: Collaborative Product Development

In this section, we introduce a simple application scenario which we shall use throughout the paper to illustrate the basic concept. The scenario is a simplified version of a more complex application piloted in the context of the ATHENA IP[4]. The scenario describes the the interaction between an Automotive Original Equipment Manufacturer (OEM) and a large supplier network consisting of first-, second- and nth-tier suppliers, in the process of Strategic Sourcing. Strategic sourcing is an early step within Cooperative Product Development (CPD), where Original Equipment Manufacturers set up strategic partnerships with the larger (so-called first-tier) suppliers with the aim of producing specific subsystems (e.g., powertrain, safety electronics) of a planned car series [4]. In the use case considered for this paper, the Original Equipment Manufacturer (OEM) issues Requests for Quotations (RfQs) to its first-tier supplier (1). The first-tier supplier serves as a gateway to the supplier network; it reviews the specification and negotiates conditions with second-tier suppliers (2) and checks the feasibility of the request. After that, the first-tier supplier issues a quote or suggests changes to the OEM (3). This cycle is repeated until all parties agree on a feasible specification. Finally, the quote is sent from the first-tier supplier to the OEM.

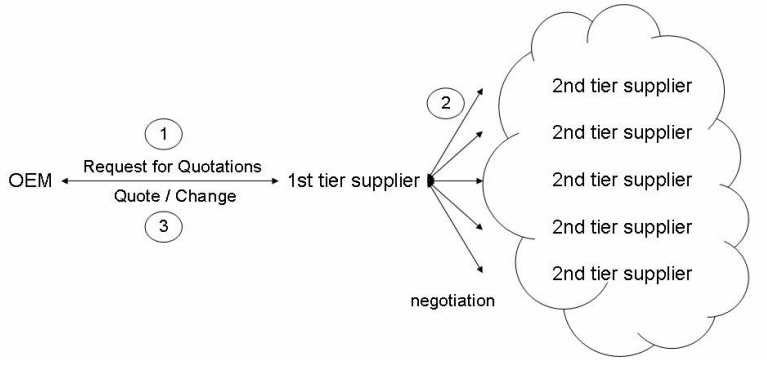


Fig. 1. Use Case Scenario Collaborative Product Development (CPD)

[4] http://www.athena-ip.org

On the OEM side we find a process-driven environment, implemented using a standard collaborative process execution engine. On the supplier side, we find second-tier suppliers joining and leaving the environment very dynamically. This results in an event-driven collaboration paradigm being appropriate for communication and collaboration in the supply network, as opposed to the process driven paradigm used for communication between the OEM and the (larger) first-tier suppliers. In the remainder of this paper, we advocate a combination of P2P and agent technologies on the supplier side, coping with the requirements resulting from the dynamic, event-driven environment.

3 Background

3.1 BDI Agents for Business Process Modeling and Enactment

Business process management (BPM) is an application domain that has already attracted significant interest of researchers in the area of MAS. Widely recognized work on agent-based BPM was done for example in the ADEPT project [5,6]. The importance and the level of recognition BPM has been receiving in the MAS community is also reflected by a panel titled "Business Process Management: A Killer App for Agents?" held at International Joint Conference on Autonomous Agents and Multiagent Systems in 2004.

What makes multiagent systems appealing both from a modeling and a runtime angle is their ability to provide natural mappings to concepts usually found in BPM models, such as organizations, roles, and goals, and their intrinsic support to flexible business service composition and loosely coupled coordination and cooperation, as they are often found in collaborative, cross-enterprise business environments [7]. Especially BDI agents were identified as a powerful vehicle for the specification and the execution of business processes. For instance, Agentis Software offers a BPM tool that is based on BDI agents.[5] Thus, agent technologies have the potential to make BPM and business process execution more flexible. In order to unleash this potential and to make it applicable to practical business applications, model-driven development (MDD) techniques have been proposed in the literature: agent technologies are to be used as an execution platform for business process models that are defined at a platform independent level [8,9]. The underlying idea is to transform models that are provided by enterprise or business process modeling tools (such as ARIS, Maestro, or MO2GO, see [7]) into models that can be directly executed in the agent environment.

The starting point for this paper has been the observation that there are some fundamental limitations in the level of support current agent platforms can provide for the execution of business processes. In particular, popular platforms such as Jade or Jack base their interaction models on the paradigm of message-based communication between agents. This is well suitable for business processes that are clearly structured according to e.g., FIPA interaction protocols [3]. However,

[5] http://www.agentissoftware.com/

it is less suitable to model and support processes that are less structured, and more event-based.

Examples for processes that can be easily supported by today's agent-based solutions are order management or procurement processes, which are complex in that there is a wealth of situation-dependent choices regarding the behavior of the individual participants, but which are structured in a sense that interaction follows clear rules and protocols. Business standards like RosettaNet[6] are mostly focusing on this type of protocols. For instance, and coming back to the application example from Section 1, communication between the car manufacturer (OEM) and the first-tier supplier, in which the OEM distributes the RfQ and suppliers create and return bids, is a fairly well-structured business process, which lends itself to modeling using a Contract-net like protocol [10].

An instance of a less structured process in the CPD example is the collaborative revision of technical specifications of the RfQ and the joint creation of a bid by the suppliers in the supplier network. Within this process, it is not at all clear nor well-specified how long it will take to reach a mutually agreed state; partners may communicate, suggest changes, make annotations at any point in time. In this case it might be possible to describe the individual reactions to upcoming events, such as changes made (suggested) to the model by a partner. However, the sequence of the reactions will depend on the local behavior of each partner.

It is our claim that agent-based *explicit* messaging needs to be complemented by an *implicit*, blackboard-style communication and coordination paradigm to be able to support unstructured, document-centric and event-driven business processes. In the collaborative product design use case, a suitable paradigm for the interactions within the supplier network is that of a P2P organization where each party maintains copies of the documents or models in question, and is notified when another party changes one of the documents. The negotiation comes to an end when each party agrees to the current set of documents available. The idea to describe a process by reactions to changes of documents is radically different from traditional business process design.

The following subsection provides an overview of existing P2P protocols and their application in business process management. In Subsection 4 we describe our basic approach towards integrating agent and P2P methods in a unified framework, as well as related work in this area.

3.2 Peer-to-Peer Computing for Decentral Business Resource Management

Over the past five years, and initially driven by Internet file sharing applications, P2P computing has been rediscovered as a paradigm for decentral, robust, and dynamic resource management. More recently, P2P protocols were successfully used to enable decentral address lookup and network routing for IP telephony. Taking a very rough categorization, the spectrum of P2P infrastructures can be divided into two groups. One group uses some form of flooding mechanism to

[6] http://www.rosettanet.org

relay requests for information to all nodes in an attempt to discover resources provided by those nodes. Examples for this technology are the gnutella network [11] and the JXTA open source project initiated by Sun Microsystems [12]. While the flooding approach offers a flexible way to formulate information queries, it has some drawbacks regarding the network traffic that is generated by a single search request [13]. Additionally, a negative request for a certain piece of information is no guarantee that the information is not available. Instead, the peer holding the information could have been out of reach for the query which usually has a limited lifetime and outreach to keep the system scalable.

The other group of middleware realizes a distributed information space that is very similar to a distributed hash table (DHT). More strict query semantics are offered by DHT systems where a query can be assumed to yield a result if and only if a certain piece of information matching the query is stored among the nodes forming the DHT. Such a definitive answer is required for business applications built on top of such an information infrastructure. Therefore, we selected a DHT based fabric as the foundation of the BRMF.

An information space offering DHT characteristics is realized by such projects as Chord [14], Pastry [15] or Tapestry [16]. Those systems allow placing and retrieving objects based on the key associated with the object, but they do not support complex XPath queries over the contents of stored XML fragments. The Siemens-owned Resource Management Framework (RMF) developed by Siemens [17] goes beyond those basic DHT mechanisms and offers functionality for the storage, discovery, retrieval and monitoring (through subscription and notification mechanisms) of arbitrary XML resources.

To the best of our knowledge, there is no unified system combining business resource management and P2P data management, even though the lack of an intermediary *hub* for business to business communication is an argument for the adoption of a P2P model [18]. The PBiz model [19] extends the Chord routing mechanism to map XML resource descriptions onto a P2P infrastructure. There is no mapping of existing schema definitions into the PBiz system. Business users operate directly on the P2P data model instead, relying on a generic query interface. Also, no details are given on how the system should counter degeneration of the P2P index structure. A model to support business processes in a P2P marketplace scenario is presented in [20]. This work focuses on defining the meta model for market place based interaction without handling resources or a concrete realization of such a system. In [21] a layered business object oriented architecture is introduced supporting meta-modeling, object management, workflow management, directory services and communications. However, only a high level description of the required subsystems based on a a centralized registry model is given.

3.3 BRMF: The Business Resource Management Framework

In the following, we will briefly introduce the Business Resource Management Framework (BRMF), which is a P2P based business collaboration platform matching the requirements on the supplier side, and addressing some of the

above-mentioned open research issues. For a more detailed description of BRMF, we refer to [22], [23].

The BRMF implements a P2P lookup algorithm based on a distributed hash-table. From the user's point of view, the BRMF forms an abstract information space, managing shared resources. Each resource in the information space is associated with one or more keywords. Using these keywords, business partners can retrieve and modify resources. Moreover, business partners can issue subscriptions for certain keywords in order to be notified if a new resource with the same keywords is published or an existing one is changed.

Figure 2 gives an overview of the architecture of the BRMF. Moving bottom-up, one can see that the BRMF is based on the Resource Management Framework (RMF) [17], which is a generic API covering P2P overlays. The RMF exposes methods to publish, subscribe and modify resources in the P2P information space.

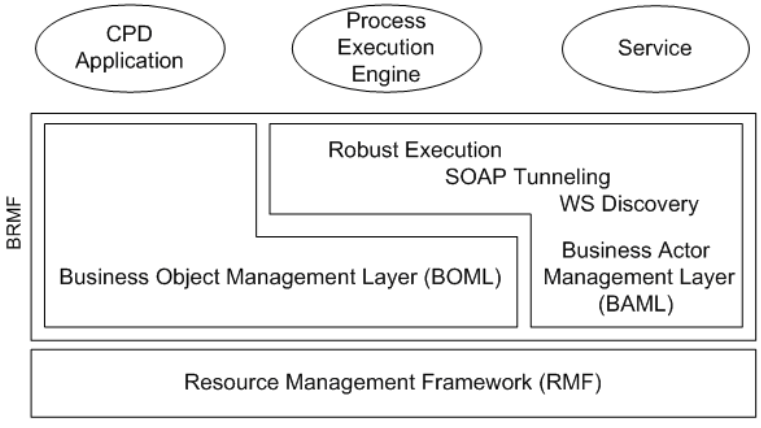


Fig. 2. Architecture of the Business Resource Management Framework (BRMF)

This infrastructure maintains the information space as an overlay network in a decentral, self-organizing way, reducing the overhead of manual configuration required when new partners join or leave the collaboration environment. The BRMF is resilient to node failure. If one of the business partners leaves the network, replicas of its resources are still kept in the information space for a certain time, allowing continuation of the overall process.

On top of the RMF, the BRMF adds explicit support for dealing with business objects. The Business Object Management Layer (BOML) provides a mapping from arbitrary XML-based business documents into RMF resources and vice versa. Moreover, the handling of keywords and XPath queries is simplified using the BOML API. The exchange of documents among several business partners can be implemented using the BRMF's publish/subscribe mechanism. Confidential communication is provided by a security layer [24]. THe third component shown

in Figure 2 is the Business Actor Management Layer (BAML) dealing with active business objects like Web Services; this component will not be used for the scope of the scenario presented in this paper.

4 Integrating Peer-to-Peer and Agent Concepts

4.1 Related Work

In this section we shall provide an overview of related work trying to bring together the concepts of agents and P2P computing. From its conception in the 1980s, multiagent systems were always perceived to be P2P organizations, i.e., collections of autonomous components capable of communication. Thus, the metaphor of a multiagent system or architecture is very much compliant with that of a P2P system or architecture. Broadly looking at the communication side, it appears that the focus in multiagent systems research is very much on higher-level aspects of coordination, cooperation, and negotiation in a goal-oriented way, whereas communication in P2P systems serves lower level decentral resource management purposes. Looking at the architecture of the individual agent, a rich body of research is available (see e.g. [25] for an overview) describing possible architectures of an agent as a proactive, re-active, and social entity.

Peer-to-peer computing on the other hand has largely focused on providing protocols for robust and scalable decentral sharing, access, and management of resources. As far as the internal structure or process logic of the individual peer are concerned, P2P systems usually do not make any assumptions beyond the necessity for a peer to implement certain interfaces e.g., for lookup operations.

Over the past few years, a number of different aspects relating agent and P2P concepts were investigated in the literature. In particular, the workshop series "Agents and Peer-to-Peer Computing" which has been held yearly since 2002 provides a good forum for articles relating these two concepts.

One general idea of using agents to add value to P2P systems is by providing semantic information on top of a P2P architecture. In [26], the usage of a semantic overlay network (SON) is proposed for the content-based routing of queries. The approach relies on a global reference ontology (classification hierarchy). Technical issues addressed in the paper are query and document classification, the Layered SON algorithm allowing a peer to decide in which Semantic Overlay Networks it wishes to be member, and strategies for searching in Layered SONs. In [27], the authors describe a layered P2P architecture for semantic search and data integration, where a network of conceptual peers (called SINodes) provide data mediation functionality. The P2P network is supported by a multi-agent system that forms the interface to the users by sharing and mapping ontologies required to perform data integration. In this work, the term P2P is used in a broad conceptual sense to describe the message-based interaction between the SINodes. It does not entail the use of P2P techniques for decentral resource management.

A second usage of agent technologies that was proposed in the literature concerns the improvement of task management and routing (in a P2P grid context),

as well as the agent-based implementation of different types of policies for managing resources in P2P systems. E.g., [28] propose the use of mobile agents to decentrally route (carry) tasks to computational nodes in a P2P grid environment. The paper provides a model for describing the short-term dynamics of task distribution as well as the long-term dynamics of task-handling. In [29] the authors propose the use of agent technology for the design and enforcement of policies to guide policies for resource sharing in virtual P2P communities, such as e.g. the Kazaa file sharing application. A formal model is provided for describing concepts such as agents, norms, resources, authorization and delegation processes.

In [30], the authors take a different stance at the added value that agents can bring to P2P computing. This overview paper raises some problem related to control, authority, and ownership in P2P systems and discusses the applicability of agent related concepts to provide solutions to these problems. In particular, the authors argue that mechanism design can be applied to P2P trading systems, that argumentation and negotiation schemas can be applied to P2P social choice problems, and that concepts of electronic institutions, norms, rules and policy languages are applicable to context management in P2P ubiquitous computing problems. Yet, this good overview paper does not make a more specific technical contribution.

To our knowledge, an area that so far has not been covered in the literature is the integration of agents and P2P computing for business-related applications, such as business process management and computer-supported collaborative work. In [31] the authors' view on the relationship between agents and P2P technologies is that the agent paradigm can be superimposed on peer architectures and that

> the agent paradigm and P2P computing are complementary concepts in that cooperation and communication between peers can be driven by the agents that reside in each peer. Agents may initiate tasks on behalf of peers.

This view is very much compliant with our approach where agents act as software entities responsible for coordination, control, decision making and interaction e.g. in cross-enterprise business process enactment and monitoring, and where an underlying P2P infrastructure is used for virtualization of business resources. In the next section, we outline our basic idea to realize this type of integration between the two paradigms.

4.2 Discussion of Main Concepts

This section describes our conceptual views on agents and P2P computing as well as a potential useful integration. To summarize, the notion of P2P computing has so far been used in two different directions, the first of which is that of a general paradigm for decentral communication, abstraction of client-server concept. In this sense, a multiagent system is a P2P system.

The second, more specific direction views a P2P system as a class of algorithms and protocols for decentral resource management. This second direction is our view on P2P computing. The focus of this view is on sharing (virtualization), synchronization, and discovery of documents, information objects, and services. Taking this stance, there are clear differences and even complementarity between agent/multiagent concepts and P2P systems.

Firstly, while agents traditionally communicate using explicit (asynchronous) messaging (e.g., based on FIPA interaction protocols [3], P2P systems such as the Business Resource Management Framework discussed in Section 3.3 create a virtual P2P information space, that can act as a kind of blackboard systems enabling implicit and event-based communication and synchronization. As outlined in the introduction, we believe that the combination of these two communication styles can provide an immense value for numerous business applications by supporting process-driven as well as event-driven, document-centric environments, and their integration.

Secondly, as quoted by [31] above, the ability of agents to reveal flexible behavior, and to provide methods for coordination, negotiation, and policies for efficient resource allocation e.g. by means of market mechanism design [30] complements the ability of P2P protocols such as Chord [14] or Kademlia [32] to provide efficient, scalable, and truly decentralized resource management features that have been proven in large-scale applications – which notably sets P2P systems apart from the level of achievement reached by today's FIPA compliant agent platforms. Thus, agents and multiagent systems can clearly benefit from P2P technologies.

In contrast, P2P platforms can also benefit from functionality provided by agents in a number of ways. One way is the ability to use agents in order to express behavior within a computational peer. In a business context, an agent can maintain and enact a business process model. Another aspect to be considered is the use of agents to introduce semantic models for dynamic and automated service selection and service composition. Also, agents can add to a P2P platform the ability for explicit and asynchronous, message- and speech-act based communication, as well as negotiation capabilities and team coordination support.

In this work, we propose to combine the use of robust decentral P2P indexing structures, such as Distributed Hashtable for resource discovery and resource sharing with an agent-based layer responsible for knowledge representation, business process enactment and monitoring.

In particular, we start from a specific class of agents, Belief-Desire-Intention (BDI) agents ([33]. As a first step to integrating BDI agents with P2P resource management, we shall present a concept to allow BDI agents to make (i.e., *virtualize*) well-defined sections of their belief base accessible to other agents (e.g. in a team), and to synchronize their beliefs by means of a P2P infrastructure. This concept – which we call *belief base virtualization*, as well as an outline of its implementation, and an example of its use are described in the following sections.

5 Belief Base Virtualization: Concept

5.1 Basic Concepts

BDI Agents. As discussed in Section 4, our approach to integrate P2P and agent technologies is to view the P2P network as a shared information or probably more appropriate knowledge space. Knowledge in an agent is represented and stored within a belief base. BDI (belief, desire, intention) agents are the most widely discussed type of agents when reasoning about knowledge and action in this context is investigated. A straightforward idea to integrate the knowledge in the P2P network with BDI agents is to maintain within an agent a set of beliefs that represents the shared knowledge. Then, each agent acts as a peer in a BRMF P2P network (see Section 3.3).

The communication in BRMF is implemented via the BRMF's publish / subscribe mechanism. In order to subscribe a keyword, participants in the BRMF issue a subscription resource for that keyword. Whenever a resource is published in the BRMF, the peer being responsible for that resource checks the availability of subscription resources with the same keyword. For each subscription, the issuers of that subscription is notified.

This yields two requirements for the implementation of a shared belief base. First, publish/subscribe implements a one-to-many communication, where the sender does not know exactly how many recipients are subscribed for a message. Second, search functions are bound to keywords. Complex subscriptions, like ontological queries, are not yet supported by the BRMF registrar component.

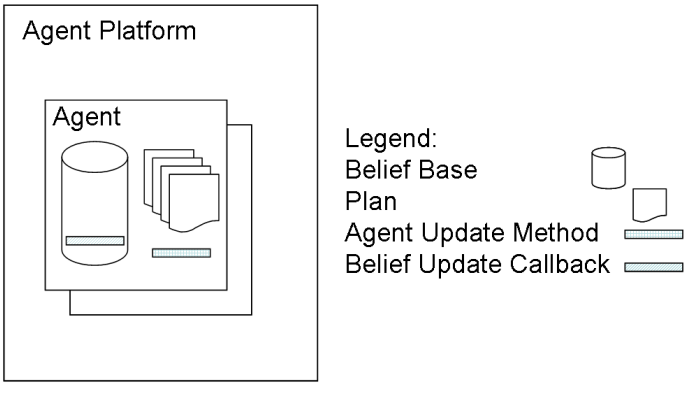


Fig. 3. Basic Building Blocks of BDI Agents

Figure 3 explains the basic structure of a BDI Agent. Because we want to be as concrete as possible we refer to Jack Intelligent Agents as a development tool for BDI agents and explain our approach with the concepts that are available in Jack. The building block of a BDI agent are the agents belief base and a library

of plans. The third concepts to mention are events. The idea is that an BDI agent is able to react to a set of predefined events and it does so by checking the plan library to look for a plan that is relevant for the event. Additionally to the relevance condition, which puts constraints on the events a specific plan might react to, one can specify a context condition which needs to evaluate to true to render the plan applicable as a reaction to a specific event. The context condition is a logical expression, which most likely extracts information from the agent's belief base. The beliefs are structured according to a relational data model. This means that each belief set forms a relation which is defined by its name and a set of attribute value pairs where the values might be of a simple type, e.g. int, float, string, etc., or the complex type Object. For each belief set a set of queries can be specified where simple queries are used to basically extract information from existing belief set elements and complex queries are logical combinations of simple queries. A simple query can for example ask whether a belief element that matches a specific pattern is present in the belief set or not. If the belief set is defined to be closed world the answer to the query evaluates to true in case a belief is present that matches the pattern specified in the query otherwise the query evaluates to false. In case the belief set is specified to be of type open world, the result is undefined in the case that no belief is present that matches the specified pattern.

BRMF integration requirements. In Section 3.3 we introduced the BRMF as a P2P based business collaboration platform, being self-organizing, and resilient to node failure. In this section, we will focus on the metamodel offered by the BRMF implementation and highlight the requirements when implementing a virtualized belief base on top of the BRMF.

As shown in Section 3.3, each resource in the BRMF is associated with one or more keywords. Access to the resources is implemented through the BRMF's registrar interface. There are several kinds of registrars. For the purpose of implementing a sharable belief base, we focus on the strict registrar, which is an implementation of the standard lookup in distributed hashtables. The strict registrar offers exact searches for known keywords. It is still an open research topic to implement more generic registrars, enabling searches for semantically rich queries, like ontological queries.

The communication in BRMF is implemented via the BRMF's publish / subscribe mechanism. In order to subscribe a keyword, participants in the BRMF issue a subscription resource for that keyword. Whenever a resource is published in the BRMF, the peer being responsible for that resource checks the availability of subscription resources with the same keyword. For each subscription, the issuers of that subscription is notified.

This yields two requirements for the implementation of a sharable belief base. First, publish/subscribe implements a one-to-many communication, where the sender does not know exactly how many recipients are subscribed for a message. Second, communication is bound to keywords. Complex subscriptions, like ontological queries, are not supported.

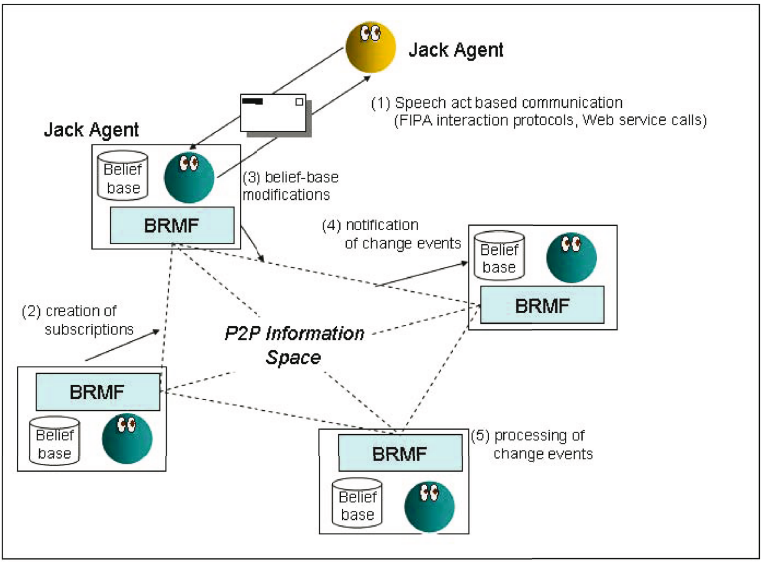


Fig. 4. Overall integration architecture

5.2 Conceptual Architecture

Figure 4 illustrates the basic conceptual integration of the agent platform with the BRMF. Agents can communicate with each other in two ways: By exchanging messages, and via virtualized shared objects maintained in the P2P information space (1). The latter form of communication is employed to implement belief base virtualization.

First, an agent will decide to share some parts of a belief base with other agents. In doing so, it will use the BRMF `publish` function to publish the corresponding belief sets (step 2 in Figure 4). Other agents can now access the virtualized belief base using the BRMF `search` method and automatically receive change events on the belief sets by means of the BRMF `subscribe` method (step 3). In case the owner of the published belief base changes the belief, all subscribed agents will receive change events as callbacks via the BRMF (step 4). Agents can then start event handler routines to initiate appropriate action. This way, the P2P information space allows agents to synchronize certain parts of their belief bases in an efficient and elegant manner.

As we shall see in the following section, the interaction between the BRMF and the Jack agent framework is implemented using a wrapper class which makes the BRMF accessible to a Jack agent and which can perform callbacks on the Jack agent. From the BRMF's point of view, the wrapper class is an application using the standard BRMF interface. The wrapper implementation is described in detail in Section 6.

6 Belief Base Virtualization: Implementation

This section outlines an implementation of the concept of belief base virtualization described in Section 5. We discuss the implementation based on the BDI agent platform Jack and the P2P platform BRMF (Business Resource Management Framework).

6.1 The Jack Agent Platform

Jack Intelligent Agents^{TM} is one of the most sophisticated tools for the design of BDI agents. The tool was developed by Agent Oriented Software[7]. In this paper, we concentrate on the concepts that are relevant for the paper and refer to the Jack documentation which is available from the Web pages for further details.

In Jack agents are composed of components: belief data, capabilities, events, and plans[8]. In the context of using a P2P middleware to build a shared belief space among different agents, the agents' belief data is the most prominent concept that is of interest for this paper. Jack takes a relational approach to structure the beliefs of an agent. Each belief set is a relation with a name—most likely the name of the domain concept the beliefs in the belief set should represent—an a list of fields, i.e. attributes of the domain concepts. Fields have types which specify which kind of information can be stored in a specific field. Fields can have simple types like for example boolean, char, float, int etc. If a more complicated structure is to be stored in a field, the generic type Object must be used. The idea is to store documents from BRMF as objects in a field of a belief set. For publishing these objects, it is very helpful that Jack allows to specify callbacks which are called automatically, as individual beliefs are created or modified in the belief set. This means that it is possible to allow the agent to read and manipulate the data in the belief set without having to deal with the underlying synchronization of the changes with the information in the BRMF.

However, only information that is entered into the belief set by the agent can be passed on to the BRMF by the callbacks. If information is changed in the BRMF other concepts are needed to make it available for the agents. Each agent has to register with BRMF specifying what information it wants to subscribe to. In case an object is changed to which a specific agent did subscribe, BRMF invokes a callback method in the agent's context, passing the changed document as a parameter. In the context of the callback method that belongs to the agent it is possible to post an event to which the agent reacts with one of its plans. An event can be seen something similar to an internal message the agent can use to pass information from one piece of code to another. Events are specified by event types. Different types of events can cause different reactions regarding specific plans. In the context of this paper events are basically used to pass on the information from the BRMF to the agents belief base. For this

[7] http://www.agent-software.com.au

[8] Interesting enough, none of these components are mandatory.

the most simple event type is sufficient. Whether more complicated event types like for example *BDIGoalEvents* can be usefully applied in this application will depend on the application domain. The event that is posted by the method called from BRMF carries the changed document as an object in one of its fields.

The plan that reacts to the event posted by the method called from the BRMF can first of all check on the object that represents the modified document. It can further span off activities within the agent as a reaction to the changed documents. Eventually, the updated object should of course find its way into the belief set that takes up all objects, i.e., documents, that are shared between the BRMF and the agents.

6.2 Implementation of a Shared Belief Base

In Section 5 we analyzed the requirements when implementing a shared belief base on top of the BRMF. We learned that BRMF enables communication via publishing and subscribing resources. In this section, we will show how such a virtualized belief base can be represented in the BRMF information space, and how notifications in the BRMF are mapped to events in the Jack agent platform.

The virtualized belief base is a set of beliefs, just as BRMF information space is a set of resources. Thus, it is straightforward to implement the elements of the shared belief set as resources in the information space. Beliefs in Jack are tuples consisting of one or more keys and one or more values. When the beliefs are transformed into BRMF resources, the keys are used as keywords in the BRMF. The resources are published once for each keyword. The payload of the resources consists of the complete list of all keywords and all values. That way, an original Jack belief can be restored when a copy of a resource is retrieved from the BRMF information space.

While Jack beliefs can be implemented one-to-one as resources in the BRMF, it is not possible to find a one-to-one mapping between Jack's events and BRMF notifications. In the BRMF, there are only three kinds of notifications: Add, Update, and Remove. These notifications are available for each keyword subscribed. In Jack, it is possible to implement arbitrary custom event types. When integrating Jack and BRMF, this feature must be restricted, and only the BRMF's notifications can used as Jack events. That means, that an Agent's plan must only handle an Add, Update, and Remove event for each subscribed keyword.

6.3 Bootstrapping

The best place to put the initialization code for BRMF for each agent is the agent's constructor method. As a first step, the agent has to make sure that the local agent platform is part of the P2P information space by calling the BRMF `init` method. In the constructor method the binding between an agent and a BRMF wrapper object are constructed, resulting in the following general constructor of the `BRMFWrapper`.

```
public BRMFWrapper(Agent a){
  BRMF.init();
  this.agent=a;
  agent.setBRMFWrapper(this);
}
```

However, the agent does not receive any documents unless it subscribes for them in the BRMF. To access the content of documents of BRMF the agent needs know know the structure of these documents; as explained in Section 6.1, these structures are introduced into the agent model by a dictionary and external classes that represent the structures in the dictionary. However, this information as well as the specifications which documents the agent should subscribe to and how to react to changes to these documents establish domain-specific knowledge. An appropriate way to specify this information within an agent is to store it into two types of plans in the plan library: plans that appropriately subscribe to BRMF documents, and plans that specify the reaction to specific updates of specific documents. Therefore, the BRMF wrapper provides a method to subscribe to new documents in the P2P information space and implements the appropriate event handler interface to receive such resources and notify the agent:

```
public void subscribe(Sring[] keywords,String structuralQuery){
  BRMF.subscribe(keywords,structuralQuery,this);
}
public void onEvent(Event e){
  ...
}
```

While the subscription actions might very well be part of the specification of a business process, the reactions to the changes are not likely to follow the usual sequencing of a business process and is therefore difficult to represent directly on the level of the business process. The better view to this is to regard the individual plans in this behavior description as stimulus-response patterns to individual update events coming from the BRMF.

If we use one belief set type with the structure (BRMF key: integer value: Object) we first of all have to define this belief set type together with all callbacks for creation, update and deletion of facts. The belief set type can then be introduced as named data in any desired agent.

6.4 Event Creation and Propagation

In the following, the integration of BRMF and the Jack agent platform regarding the creation and propagation of events are described. Previously, the `BRMFWrapper` was introduced as the component bridging the P2P system and the agents in the agent platform. Two basic actions of the agents are distinguished: publishing a new belief into the shared belief base and updating an existing belief. For each of the actions there are two directions of information flow from the wrapper perspective. Either the wrapped agent publishes or

updates a belief, and the new belief must be propagated as a new resource in the P2P information space. In the other direction, a new or updated resource becomes available in the information space, and the wrapped agent must be notified of the new belief.

Each BRMF resource object held by the wrapper is associated with a unique identifier, used in interactions between the agent and the wrapper. Both event types are handled in the wrapper's `onEvent` method that was registered with the BRMF and the internal identifier for the resource is used in the interaction between the agent and the wrapper.

```
public void onEvent(Event e){
  Resource r=e.getResource();
  if (e.isPublishEvent()){
    Identifier i=createIdentifier(r);
    agent.addBelief(i,r.getContent());
    r.addUpdateListener(this);
  } else if (e.isUpdateEvent()){
    Identifier i=getIdentifier(r);
    agent.updateBelief(i,r.getContent());
  }
}
```

To enable explicit propagation of updates or new beliefs by the agent, corresponding `publish` and `update` methods are provided by the wrapper.

```
public void update(Identifier i,String content) {
  Resource r = lookupResource(i);
  r.setContent(content);
  r.commitUpdates();
}

public Identifier publish(String content) {
  Resource r = new Resource();
  r.setContent(content);
  Identifier i = createIdentifier(r);
  return i;
}
```

Currently, the update method is explicitly limited to the original creator of a resource, concurrent update of the resource is not supported. This meets the requirements of our use case, which will be discussed in the next section.

7 Application: Automotive Collaborative Product Development

This section demonstrates the integration of Jack and BRMF for belief set virtualization using the example application of Automotive collaborative product development (CPD) introduced in Section 1.

In the use case scenario, the first-tier and second-tier suppliers are each represented by Jack agents. The communication between the agents is implemented via a virtualized belief base shared through the BRMF information space. In this section we will show how the interactions in the use case are mapped into events in the combined agent and P2P environment.

A new collaborative product design phase is started when the first tier supplier receives a RfQ. The first-tier supplier subsequently publishes sub-RfQs into the P2P information space, for each material to be ordered by second-tier suppliers. The second-tier suppliers are subscribed for their respective keywords and are notified of the `publish` event in the BRMF. The agent's BRMF wrapper adds the resource into the agent's belief set. This triggers the execution of a plan in the agents, which is the starting point for the internal business process of the second-tier supplier.

Finally the second-tier suppliers publish Quote documents into the shared belief base, upon which the first-tier supplier is subscribed. These quote documents may contain change requests to the technical specifications. In reaction to the Quote the first-tier supplier either accepts the quote or updates the RfQ resource. This cycle is repeated until the first-tier supplier settles for a quote.

As an example interaction, a first-tier supplier might publish an RfQ for an Air Bag Inflator made of steel.

```
<RawMaterials>
  <SingleMetallicRawMaterials>
    <MaterialName>steel</MaterialName>
    <MaterialSpecification>steel</MaterialSpecification>
    <InterestedSubPart>AirBag Inflator</InterestedSubPart>
  </SingleMetallicRawMaterials>
</RawMaterials>
```

Now assume that a second-tier supplier from the metallic industry is subscribed for parts in the automotive industry made of steel, and is thus notified about the new quote. The sales engineer of the second-tier supplier processes the quote and proposes magnesium as an alternative material. This alternative material is included in the quote published by the second-tier supplier.

```
<Alternative>
  <SingleMetallicRawMaterials>
    <MaterialName>carbon</MaterialName>
    <MaterialSpecification>carbon</MaterialSpecification>
    <InterestedSubPart>AirBag Inflator</InterestedSubPart>
  </SingleMetallicRawMaterials>
</Alternative>
```

In reaction to this, the first-tier supplier may accept the alternative material and update its RfQ with a new material specification.

After the technical negotiation phase is finished and all partners agreed on a specification, the business partners start a second negotiation phase negotiating the price of the part.

At the agent's side, the structure of the documents which are passed on between the agents and BRMF is described by a Jack dictionary. The dictionary entries corresponding the the above fragments of BRMF documents are

```
<Class :name "RawMaterials"
       :fields (
        <Field :name "SingleMetallicRawMaterials"
          :type :class :subtype "ST_SingleMetallicRawMaterials">
       )
>
<Class :name "ST_SingleMetallicRawMaterials">
       :fields (
       <Field :name "MaterialName" :type :string>
       <Field :name "MaterialSpecification" :type :string>
       <Field :name "InterestedSubPart" :type :string>
       )
>
```

and

```
<Class :name "Alternative"
       :field (
       <Field :name "SingleMetallicRawMaterials"
         :type :class :subtype "ST_SingleMetallicRawMaterials">
       )
>
```

The structures for the dictionary entries can be directly extracted from a XML schema definition in an automated manner. Although Jack supports the serialization and deserialization of its internal objects into XML documents, it is unfortunately not the case that this serialization would directly correspond to the XML documents directly derived from the XML schema specification. One needs to apply an XSLT transformation to convert one into each other to guarantee interoperability.

The type descriptions in the Jack dictionary are introduced as external classes in Jack which means that the specified types are directly accessible for the agents. Corresponding to the concrete instances specified above an agent could use

```
RawMaterial rm = new RawMaterial();
rm.SingleMetallicRawMaterials.MaterialName = "steel";
rm.SingleMetallicRawMaterials.MaterialSpecification = "steel";
rm.SingleMetallicRawMaterials.InterestedSubPart = "AirBag Inflator";
```

and

```
Alternative alt = new Alternative();
alt.SingleMetallicRawMaterials.MaterialName = "carbon";
alt.SingleMetallicRawMaterials.MaterialSpecification = "carbon";
alt.SingleMetallicRawMaterials.InterestedSubPart = "AirBag Inflator";
```

The agent either asserts the object as a belief set entry into its BRMF belief set from which it is automatically published to BRMF from the belief set's update callback or the agents gets the object from BRMF where BRMF calls the agents update method which send the agent an event that contains the object. In the latter case it is in the responsibility of the plan that reacts to the incoming event to enter the object as a belief set entry into the agent's BRMF belief set. However, this gives the agent the chance to reject updates of objects that come from BRMF risking inconsistency of local beliefs regarding the state of corresponding objects in BRMF.

8 Discussion and Outlook

The main contribution of this paper is the description a concept (including a proof-of-concept implementation) for an integration of multiagent and P2P concepts for business process management application. The concept provides approaches for (1) integrating process-centric with event-centric modeling at the business process level; (2) extending "explicit" message-based agent communication by implicit communication based on shared objects at the technical level; and (3) integrating P2P protocols with a BDI agent architecture to enable belief set virtualization across agents, agent teams, and agent platforms.

To our knowledge, the paper constitutes the first published work on combining agent and P2P concepts for business process management. Topics for future work (see Section4 for a more detailed description of related work). It focuses on a single aspect of decentral resource management and implicit communication within and across a multiagent system. Future work will examine other aspects where agent and P2P computing can be usefully integrated. One prominent area is that of modelling support, including applying model-driven development concepts to automatically derive agent and P2P implementations starting from a business level specification of the required process / behavior. This will in particular require more work at the P2P side to refine the metamodel and provide necessary model transformations to create BRMF models directly from e.g., a PIM4SOA representation [34], as well as extensions to the current Jack metamodel to create the necessary callbacks from a PIM-level P2P model. Another area of concern lies in distributed transaction control issues, i.e., how to maintain ACID properties on belief sets. In this respect, the current approach relies on a simplifying restriction imposed by the BRMF, such that every published resource has an owner and can only be modified via the owner. Lifting this restriction will make transaction control mechanisms very important.

Finally, an area of future research will be to study extending the ideas of belief base virtualization to other artifact of a BDI agent's internal state to provide more powerful support of MAS concepts such as shared goals or intentions.

Acknowledgments

Part of the work reported in this paper is funded by the E.C. within the ATHENA IP under the European grant FP6-IST-507849. The paper does not represent the view of the E.C. nor that of other consortium members, and the authors are responsible for the paper's content.

References

1. Müller, J.P., Bauer, B., Friese, T., Roser, S., Zimmermann, R.: Software agents for electronic business: Opportunities and challenges (2005 re-mix). In Chaib-Draa, B., Müller, J.P., eds.: Multi-agent-based supply chain management. Studies in Computational Intelligence. Springer-Verlag (2006) 63–102
2. Luck, M., McBurney, P., Shehory, O., Willmott, S., eds.: Agent Technology: Computing as Interaction. A Roadmap for Agent-Based Computing. AgentLink (2005)
3. Bauer, B., Müller, J.P., Odell, J.: Agent UML: A formalism for specifying multiagent software systems. International Journal of Software Engineering and Knowledge Engineering (IJSEKE) **11** (2001) 207–230
4. Stäber, F., Müller, J.P., Sobrito, G., Bartlang, U., Friese, T.: Interoperability challenges and solutions in automotive collaborative product development (2007) Submitted to 3rd International Conference on Interoperability for Enterpris Software and Applications (I-ESA'2007).
5. Jennings, N.R., Faratin, P., Norman, T.J., O'Brien, P., Odgers, B.: Autonomous agents for business process management. Int. Journal of Applied Artificial Intelligence **14** (2000) 145–189
6. Jennings, N.R., Faratin, P., Norman, T.J., O'Brien, P., Odgers, B., Alty, J.L.: Implementing a business process management system using ADEPT: A real-world case study. Int. Journal of Applied Artificial Intelligence **14** (2000) 421–465
7. Greiner, U., Lippe, S., Kahl, T., Ziemann, J., Jaekel, F.W.: Designing and implementing cross-organizational business processes - description and application of a modelling framework. In: Enterprise Interoperability: New Challenges and Approaches, Springer-Verlag (2007) To Appear
8. Hahn, C., Madrigal-Mora, C., Fischer, K., Elvesæter, B., Berre, A.J., Zinnikus, I.: Meta-models, models, and model transformations: Towards interoperable agents. In: Multiagent System Technologies, Proc. of the 4th German Conference MATES 2006, Erfurt, Germany, LNAI 4196, Springer-Verlag, September 2006. (2006) 123–134
9. Ziemann, J., Ohren, O., Jaekel, F.W., Kahl, T., Knothe, T.: Achieving enterprise model interoperability applying a common enterprise metamodel. In: Enterprise Interoperability: New Challenges and Approaches, Springer-Verlag (2007) To Appear
10. Smith, R.G.: The contract net protocol: High-level communication and control in a distributed problem solver. IEEE Transactions on Computers **C-29** (1980) 1104–1113

11. Wikipedia: Entry on Gnutella (2006) http://en.wikipedia.org/wiki/Gnutella.
12. Gong, L.: JXTA: A Network Programming Environment. IEEE Internet Computing **5** (2001) 88–95
13. Ritter, J.: Why Gnutella Can't Scale. No, Really (2001) http://www.darkridge.com/~jpr5/doc/gnutella.html.
14. Stoica, I., Morris, R., Karger, D., Kaashoek, M.F., Balakrishnan, H.: Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In: Proceedings of the ACM SIGCOMM '01 Conference. (2001)
15. Rowstron, A., Druschel, P.: Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In: Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware), Heidelberg, Germany (2001) 329–250
16. Zhao, B.Y., Kubiatowicz, J., Joseph, A.: Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing. Technical report, UCB/CSD-01-1141 (2001)
17. Rusitschka, S., Southall, A.: The resource management framework: A system for managing metadata in decentralized networks using peer-to-peer technology. In: Agents and Peer-to-Peer Computing. Volume 2530 of Lecture Notes in Computer Science., Springer (2003) 144–149
18. Bussler, C.: P2P in B2BI. In: Proceedings of the 35th Annual Hawaii International Conference on System Sciences. (2002) 302
19. Chen, S., Wu, Z., Zhang, W., Ma, F.: PBiz: An E-business Model Based on Peer-to-Peer Network. In: Proceedings of Grid and Cooperative Computing, Second International Workshop, Shanghai, China (2003) 404–411
20. Schmees, M.: Distributed digital commerce. In: Proceedings of the 5th international conference on Electronic commerce, ACM Press (2003) 131–137
21. Karakaxas, A., Zografos, V., Karakostas, B.: A Business Object Oriented Layered Enterprise Architecture. In: Proceedings of the 11th International Workshop on Database and Expert Systems Applications. (2000) 807
22. Friese, T., Müller, J.P., Freisleben, B.: Self-Healing Execution of Business Processes Based on a Peer-to-Peer Service Architecture. In: Proceedings of the 18th International Conference on Architecture of Computing Systems. Volume 3432 of Lecture Notes in Computer Science., Springer-Verlag (2005) 108–123
23. Friese, T., Müller, J., Smith, M., Freisleben, B.: A robust business resource management framework based on a peer-to-peer infrastructure. In: Proc. 7th International IEEE Conference on E-Commerce Technology, IEEE Press (2005) 215–222
24. Stäber, F., Bartlang, U., Müller, J.P.: Using Onion Routing to Secure Peer-to-Peer Supported Business Collaboration. In Cunningham, P., Cunnigham, M., eds.: Exploiting the Knowledge Economy: Issues, Applications and Case Studies. Volume 3., IOS Press (2006) 181–188
25. Müller, J.P.: The design of intelligent agents. Volume 1177 of Lecture Notes in Artificial Intelligence. Springer-Verlag (1996)
26. Crespo, A., Garcia-Molina, H.: Semantic overlay networks for P2P systems. In Moro, G., Bergamaschi, S., Aberer, K., eds.: Agents and Peer-to-Peer Computing. Volume 3601 of Lecture Notes in Artificial Intelligence., Springer-Verlag (2005) 1–13
27. Bergamaschi, S., Fillottrani, P.R., Gelati, G.: The SEWASIE multi-agent system. In Moro, G., Bergamaschi, S., Aberer, K., eds.: Agents and Peer-to-Peer Computing. Volume 3601 of Lecture Notes in Artificial Intelligence., Springer-Verlag (2005) 120–131

28. Jin, X., Liu, J., Yang, Z.: The dynamics of peer-to-peer tasks: An agent-based perspective. In Moro, G., Bergamaschi, S., Aberer, K., eds.: Agents and Peer-to-Peer Computing. Volume 3601 of Lecture Notes in Artificial Intelligence., Springer-Verlag (2005) 173–184
29. Boella, G., van der Torre, L.: Permission and authorization in policies for virtual communities of agents. In Moro, G., Bergamaschi, S., Aberer, K., eds.: Agents and Peer-to-Peer Computing. Volume 3601 of Lecture Notes in Artificial Intelligence., Springer-Verlag (2005) 86–97
30. Willmott, S., Puyol, J.M., Cortés, U.: On exploiting agent technology in the design of peer-to-peer applications. In Moro, G., Bergamaschi, S., Aberer, K., eds.: Agents and Peer-to-Peer Computing. Volume 3601 of Lecture Notes in Artificial Intelligence., Springer-Verlag (2005) 98–107
31. Moro, G., Ouksel, A.M., Sartori, C.: Agents and peer-to-peer computing: A promising combination of paradigms. In Moro, G., Koubarakis, M., eds.: Agents and Peer-to-Peer Computing. Volume 2530 of Lecture Notes in Artificial Intelligence., Springer-Verlag (2003) 1–14
32. Maymounkov, P., Mazieres, D.: Kademlia: A peer-to-peer information system based on the xor metric. In: Peer-to-Peer Systems: Proceedings of the 1st International Workshop on Peer-to-Peer Computing (IPTPS02). Volume 2429 of Lecture Notes in Computer Science., Springer-Verlag (2002) 53–65
33. Rao, A.S., Georgeff, M.P.: Modeling rational agents within a BDI-architecture. In Allen, J., Fikes, R., Sandewall, E., eds.: Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning (KR'91), Morgan Kaufmann publishers Inc.: San Mateo, CA, USA (1991) 473–484
34. Benguria, G., Larrucea, X., Elvesaeter, B., Neple, T., Beardsmore, A., Friess, M.: A platform independent model for service-oriented architectures. In: Proc. 2nd International Conference on Interoperability of Enterprise Software and Applications (I-ESA'06), Springer-Verlag (2007) To appear

Part I

Asimovian Multiagents: Applying Laws of Robotics to Teams of Humans and Agents

Nathan Schurr[1], Pradeep Varakantham[1], Emma Bowring[1], Milind Tambe[1], and Barbara Grosz[2]

[1] Computer Science Department, University of Southern California
Los Angeles, California
{schurr,varakant,bowring,tambe}@usc.edu
[2] Harvard University, Maxwell-Dworkin Laboratory, Room 249
33 Oxford Street, Cambridge, MA 02138
grosz@eecs.harvard.edu

Abstract. In the March 1942 issue of "Astounding Science Fiction", Isaac Asimov for the first time enumerated his *three laws of robotics*. Decades later, researchers in agents and multiagent systems have begun to examine these laws for providing a useful set of guarantees on deployed agent systems. Motivated by unexpected failures or behavior degradations in complex mixed agent-human teams, this paper for the first time focuses on applying Asimov's first two laws to provide behavioral guarantees in such teams. However, operationalizing these laws in the context of such mixed agent-human teams raises three novel issues. First, while the laws were originally written for interaction of an individual robot and an individual human, clearly, our systems must operate in a team context. Second, key notions in these laws (e.g. causing "harm" to humans) are specified in very abstract terms and must be specified in concrete terms in implemented systems. Third, since removed from science-fiction, agents or humans may not have perfect information about the world, they must act based on these laws despite uncertainty of information. Addressing this uncertainty is a key thrust of this paper, and we illustrate that agents must detect and overcome such states of uncertainty while ensuring adherence to Asimov's laws. We illustrate the results of two different domains that each have different approaches to operationalizing Asimov's laws.

1 Introduction

Recent progress in the agents arena is bringing us closer to the reality of multiagent teams and humans working together in large-scale applications [3,4,10,11,12]. In deploying such multiagent teams and making them acceptable to human teammates, it is crucial to provide the right set of guarantees about their behavior. The unanswered question is then understanding the right set of guarantees to provide in such teams.

R.H. Bordini et al. (Eds.): ProMAS 2006, LNAI 4411, pp. 41–55, 2007.

In this paper, we focus on Asimov's three laws of robotics from his science-fiction stories that provide us a starting point for such behavior guarantees. We do not claim that these laws are the only or best collection of similar rules. However, the laws outline some of the most fundamental guarantees for agent behaviors, given their emphasis on ensuring that *no harm* comes to humans, on obeying human users, and ensuring protection of an agent. Indeed, these laws have inspired a great deal of work in agents and multiagent systems already [14,4,8]. However, in operationalizing these laws in the context of multiagent teams, three novel issues arise. First, the key notions in these laws (e.g. "no harm" to humans) are specified in very abstract terms and must be specified in concrete terms in implemented systems. Second, while the laws were originally written for interaction of an individual robot and an individual human, clearly, our systems must operate in a team context. Third, since, in many realistic domains, agents or humans may not have perfect information about the world, they must act based on these laws despite information uncertainty and must overcome their mutual information mismatch.

Indeed, as mentioned earlier, researchers have in the past advocated the use of such laws to provide guarantees in agent systems [4,8,14]. However, previous work only focused on a single law (the first law of safety) and in the process addressed two of the issues mentioned above: defining the notion of harm to humans and applying the laws to teams rather than individual agents. The key novelty of our work is going beyond previous work to consider the second of Asimov's laws, and more importantly in recognizing the fundamental role that uncertainty plays in any faithful implementation of such a law. In particular, Asimov's second law addresses situations where an agent or agent team may or may not obey human orders — it specifies that in situations where (inadvertent) harm may come to other humans, agents may disobey an order. However, in the presence of uncertainty faced either by the agents or the human user about each others' state or state of the world, either the set of agents or the human may not be completely certain of their inferences regarding potential harm to humans. The paper illustrates that in the presence of such uncertainty, agents must strive to gather additional information or provide additional information. Given that the information reduces the uncertainty, agents may only then disobey human orders to avoid harm.

To the best of our knowledge, this paper for the first time provides concrete implementations that address the three key issues outlined above in operationalizing Asimov's laws. Our implementations are focused on two diverse domains, and thus require distinct approaches in addressing these issues. The first domain is that of disaster rescue simulations. Here a human user provides inputs to a team of (semi-)autonomous fire-engines in order to extinguish maximum numbers of fires and minimize damage to property. The real-time nature of this domain precludes use of computationally expensive decision-theoretic techniques, and instead agents rely on heuristic techniques to recognize situations that may (with some probability) cause harm to humans. The second domain is that of a team of software personal assistant deployed in an office environment to assist human users to complete tasks on time. The personal

assistants face significant uncertainty in the observations they receive about the status of the human users. Here, we use partially observable markov decision problems (POMDPs) to address such uncertainty.

2 Human-Multiagent Systems

Increasingly, agents and agent teams are being viewed as assistants to humans in many critical activities and changing the way things are done at home, at office, or in a large organization. For example, as illustrated in [11], multiagent teams can help coordinate teams of fire fighters in rescue operations during disaster response. Furthermore, they are also being used as helping hand to humans in an office setting for assisting in various activities like scheduling meetings, collecting information, managing projects etc. Such a transformation seems a necessity, because it relieves humans of the routine and mundane tasks and allows them to be more productive and/or successful.

However, making such a transformation introduces many new challenges concerning how the human and agents will interact. The primary challenge that we focus on in this paper is that if humans are to trust agents with important tasks, they are going to want some guarantees on the performance of the agents. This allows the humans to be confident that problematic or dangerous situations won't arise for the humans.

Below, we will introduce two domains that have to address the challenges of including both humans and agents in real world situations. First, we will describe a disaster response simulation where a human user must help a team of fire fighter agents put out all the fires in a downtown area. Second, we present a domain where agents assist workers with assigning of duties in an office environment.

A key aspect of both domains is "adjustable autonomy" which refers to an agent's ability to dynamically change its own autonomy, possibly to transfer control over a decision to a human. Adjustable autonomy makes use of flexible transfer-of-control strategies [9]. A transfer-of-control strategy is a preplanned sequence of actions to transfer control over a decision among multiple entities. For example, an AH strategy implies that an agent (A) attempts a decision and if the agent fails in the decision then the control over the decision is passed to a human (H). An optimal transfer-of-control strategy optimally balances the risks of not getting a high quality decision against the risk of costs incurred due to a delay in getting that decision.

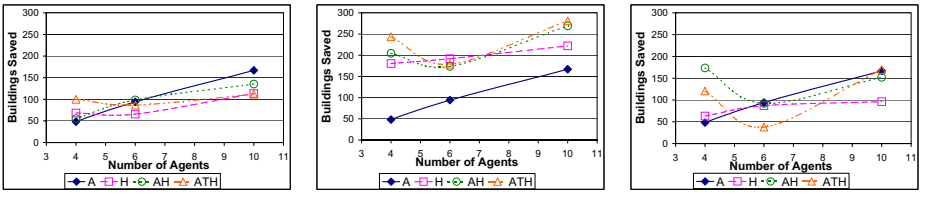
Both of these constructed systems were described in earlier publications [11,13] and had noted the problematic situations when interacting with humans. However, the diagnosis of such situations and their particular solutions are novel contributions. Because of their differing characteristics, the domains arrive at different approaches to their solutions.

2.1 Disaster Response

Techniques for augmenting the automation of routine coordination are rapidly reaching a level of effectiveness where they can simulate realistic coordination on

Fig. 1. The DEFACTO system displays multiple fires in an urban environment

the ground for large numbers of emergency response entities (e.g. fire engines, police cars) for the sake of training. Furthermore, it seems inevitable that future disaster response systems will utilize such technology for coordination among different rescue vehicles. We have constructed DEFACTO (Demonstrating Effective Flexible Agent Coordination of Teams through Omnipresence) as a high fidelity system for training and simulating of future disaster response. DEFACTO allows for a human user (fire fighter) to observe a number of fires burning in buildings in an urban environment, and the human user is also allowed to help assign available fire engines to the fires. The DEFACTO system achieves this via three main components: (i) Omnipresent Viewer - intuitive interface (see Figure 1), (ii) Proxy Framework - for team coordination, and (iii) Flexible Interaction - adjustable autonomy between the human user (fire fighter) and the team. More about DEFACTO can be found here [11].



(a) Subject 1 (b) Subject 2 (c) Subject 3

Fig. 2. Performance

The DEFACTO system's effectiveness was evaluated through experiments comparing the effectiveness of adjustable autonomy strategies over multiple users. In DEFACTO, each fire engine is controlled by a "proxy" agent [11] in order to handle the coordination and execution of adjustable autonomy strategies. Consequently, the proxy agents can try to allocate fire engines to fires in

a distributed manner, but can also transfer control to the more capable human user. The user can then allocate engines to the fires that the user has control of.

The results of our experiments are shown in Figure 2, which shows the results of subjects 1, 2, and 3. Each subject was confronted with the task of aiding fire engines in saving a city hit by a disaster. For each subject, we tested three strategies, specifically, H, AH and $A_T H$; their performance was compared with the completely autonomous A strategy. An H strategy implies that agents completely rely on human inputs for all their decisions. An A strategy is one where agents act with complete autonomy and allocate themselves to task (fires) without human assistance. An AH strategy allows the agent to possibly allocate itself to a task, and if not, then transfer control to the human, whereas the similar $A_T H$ strategy allows the whole agent team to try and allocate a task amongst the team before it will resort to transferring control to a human. Each experiment was conducted with the same initial locations of fires and building damage. For each strategy we tested, we varied the number of fire engines between 4, 6 and 10. Each chart in Figure 2 shows the varying number of fire engines on the x-axis, and the team performance in terms of numbers of building saved on the y-axis. For instance, subject 2 with strategy AH saves 200 building with 4 agents. Each data point on the graph is an average of three runs. Note that the phenomena described below ranges over multiple users, multiple runs, and multiple strategies.

Figure 2 enables us to conclude that: *Following human orders can lead to degradation in agent team performance.* Contrary to expectations and prior results, human involvement does not uniformly improve team performance, as seen by human-involving strategies performing worse than the A strategy in some cases. For instance, for subject 3, AH strategy provides higher team performance than A for 4 agents, yet at 6 agents human influence is clearly not beneficial (AH performs worse than A). Furthermore, for subject 1, following human orders leads to lower performance with 10 agents, with AH or $A_T H$, than with a fully autonomous strategy (A). We also note that the strategies including the humans and agents (AH and $A_T H$) for 6 agents show a noticeable decrease in performance for subjects 2 and 3 (see Figure 2) when compared to 4 agents. Since the performance of the fully autonomous strategy increases along with the increasing number of agents, we conclude that the culprit is the agents following human orders in AH and $A_T H$. It is very important to have the team understand which factors contributed to this phenomena and to have the team be able to prevent it.

2.2 Office Assistants

Another domain that we consider is the Task Management Problem (TMP) in personal software assistants. This is a problem that we are currently addressing as part of CALO (Cognitive Agent that Learns and Organizes), a software personal assistant project [5]. In this domain, a set of dependent tasks is to be performed by a group of users before a deadline. An example could be one where a group of users are working on getting a paper done before the deadline. Each

user is provided with an agent assistant. Each agent monitors the progress of its user on various tasks, and helps in finishing the tasks before a deadline by doing task reallocations (in case of insufficient progress) at appropriate points in time. Agents also make a decision on whom to reallocate a task, thus having to monitor status of other users who are capable of doing it. More details of TMP are in [13].

This problem is complicated as the agents need to reason about reallocation in the presence of transitional and observational uncertainty. Transitional uncertainty arises because there is non-determinism in the way users make progress. For example, a user might finish two units of a task in one time unit, or might not do anything in one time unit (a task here is considered as a certain number of units of work). Observational uncertainty comes about because it is difficult to observe exact progress of a user or the user's capability level.

Agents can ask their users about the progress made (when there is significant uncertainty about the state) or for decision on re-allocation of the current task. This asking, however, comes at a cost of disturbing the user and occurs only with a certain probability as users may or may not respond to agents request. Thus each agent needs to find an optimal strategy that guides its operation at each time step, till the deadline.

Partially Observable Markov Decision Problems (POMDPs) were used in modeling this TMP problem, owing to the presence of uncertainty. Policy in a POMDP is a mapping from "belief states" (probability distribution over the states in the system) to actions. Each user's agent assistant computes such a policy. (More information on POMDPs can be found in [6]. Though this paper does not require an in-depth understanding of POMDPs, high level familiarity with POMDPs is assumed.)

Unfortunately, within TMP an agent faithfully following orders may result in a low quality solution (low expected utility). There are scenarios where human can provide a decision input at a certain point in time, but later the agent team runs into problems because of faithfully following that decision input. For example, the human can ask the agent not to reallocate the task because of a belief that the task can be finished on time. Yet, since the human is in control of many tasks, she may not be able to finish the task on time. Also, agent faces significant uncertainty about certain factors in the domain, and hence is not in a situation to override human decisions. For example: agent can have significant uncertainty about the progress on a task, due to the transitional and observational uncertainty in the domain. The result is that:

1. *Faithfully following human decision may lead to problems*: This is because humans are in control of many tasks, and depending on the workload over time, some human decisions might need to be corrected over time. Since humans may not provide such corrections, agents may need to override the human's earlier decision.
2. *Agents need to reduce uncertainty about certain variables*: While agents must occasionally override human decisions, they face uncertainty in estimating human capability (amount of progress accomplishable in one time unit) and

information about progress. If the agent assumes a certain capability level (from previous experiences) and plans accordingly without considering human input, it might run into problems: (a) if it assumes a lower capability level, then the reallocation will happen early and (b) if a higher value is assumed, it wouldn't reallocate until very late, making it difficult for the user taking this task. Similarly, an agent faces uncertainty about actual progress on a task.

3 On Asimov's Laws

In 1942, having never had any personal contact with a robot, Isaac Asimov sat down to write his short story "Runaround" [2] and in doing so enumerated for the first time his three laws of robotics:

- *First Law: A robot may not injure a human being, or, through inaction, allow a human being to come to harm.*
- *Second Law: A robot must obey the orders given it by human beings except where such orders would conflict with the First Law.*
- *Third Law: A robot must protect its own existence as long as such protection does not conflict with the First or Second Law.*

Asimov believed these three laws were both necessary and sufficient, an idea he set out to illustrate in his series of robot stories. While he believed that correctly implemented his three laws would prevent robots from becoming the nightmarish Frankensteins that were the fodder of many science fiction stories, even Asimov admitted that the operationalization of his three laws would not be simple or unambiguous. In this paper, we focus on operationalization of the first two laws, which requires several key issues be addressed in concretely applying them to our domains of interest: (i) Providing definition of "harm" so central to the first law; (ii) Applying these laws in the context of teams of agents rather than individuals; and (iii) Addressing these laws in the presence of uncertainty in both the agent's and the human user's information about each other and about the world state. Previous work has only focused on the first law, and thus on techniques to avoid harm via agents' actions [4,8,14]. This previous work dealt with both a single agent and a team of agents, but the emphasis remained on the autonomous actions of these agents. In contrast, the second law emphasizes interactions with humans, and thus its relevance in the context of heterogeneous systems that involve both humans and multiagent teams, that are of interest in this paper.

Indeed, among the issues that must be addressed in concretely applying these laws, the first two — defining harm and applying the laws to teams instead of individuals — are addressed in previous work (albeit differently from our work). However, it is uncertainty of information that the agents and the human user may suffer from, that is the novel issue that must be clearly addressed when we deal with the second law. In the following, we provide a more detailed discussion of these three issues, with an emphasis on the issue of uncertainty. Nonetheless, in

contrast with previous work, this paper is the first (to the best of our knowledge) that addresses these three issues together in operationalizing the two laws.

3.1 Definition of Harm

What constitutes harm to a human being? Must a robot obey orders given it by a child, by a madman, by a malevolent human being? Must a robot give up its own expensive and useful existence to prevent a trivial harm to an unimportant human being? What is trivial and what is unimportant? pg 455 [2]

The notion of harm is fundamental to Asimov's laws. Yet, Asimov himself did not imply that harm to be necessarily physical harm to humans. Indeed, in the story "LIAR" harm is purely mental harm (e.g. someone not getting a promotion they wanted) [2]. So whereas the notion of harm as physical harm to humans is obviously relevant in one of our domains mentioned earlier (disaster rescue), it is also relevant in the office assistant domain, where harm may imply harm to some business (e.g. products not delivered on time) where the office assistant team is deployed. Indeed, in previous work in software personal assistants that is motivated by Asimov's laws [14,8], the notion of harm includes such effects as deletion of files or meeting cancellation.

In this paper, the notion of harm is operationalized as a "significant" negative loss in utility. So if actions cause a significant reduction in an individual agent's or team's utility, then that is considered as constituting harm. In our disaster rescue simulation domain, such negative utility accrues from loss of (simulated) human life or property. An example of this can be see when subject 3's inputs are followed by 6 fire engine agents, resulting in more buildings being burned than if the inputs were ignored. In our office assistant domain, the lack of the agent team's ability to complete tasks by deadlines provided is what constitutes harm.

3.2 Applying Laws to Teams

I have dealt entirely with the matter of the interaction between [a] single robot and various human beings. ... Suppose two robots are involved, and that one of them, through inadvertence, lack of knowledge, or special circumstances, is engaged in a course of action (quite innocently) that will clearly injure a human being – and suppose the second robot, with greater knowledge or insight, is aware of this. pg. 479-480 [2]

Diana Gordon-Spear's work on *Asimovian agents* [4] addresses teams of agents that guarantee certain safety properties (inspired by the first law above) despite adaptation or learning on part of the agent team. The key complications arise because the actions of multiple agents interact, and thus in preserving such safety property, it is not just the actions of the individual, but their interactions that must be accounted for, in terms of safety. In our work (particularly as seen in the disaster response domain of Sections 2.1 and 4.1), similar complexities arise when applying the laws to teams of agents. No single individual may be able to detect harm by itself; rather the harm may only be detectable when the team of agents is considered as a whole.

3.3 Uncertainty

Even a robot may unwittingly harm a human being, and even a robot may not be fast enough to get to the scene of action in time or skilled enough to take the necessary action. pg 460 [2]

The second law in essence requires that agents obey human orders unless such orders cause harm to (other) humans. Thus, this law opens up the possibility that the agent may disobey an order from a human user, due to the potential for harm. In many previous mixed agent-human systems including our own systems described in Section 2, human inputs are considered final, and the agent cannot override such inputs — potentially with dangerous consequences as shown earlier. Asimov's second law anticipates situations where agents must indeed override such inputs and thus provides a key insight to improve the performance of agents and agent teams.

Yet, the key issue is that both the agents and the human users have uncertainty; simply disobeying an order from a human given such uncertainty may be highly problematic. For example, the agents may be uncertain about the information state of the humans, the intellectual or physical capability of the human users, and the state of the world, etc. In such situations, agents may be uncertain about whether the current user order may truly cause (inadvertent) harm to others. It is also feasible that the human user may have given an order under a fog of uncertainty about the true world state that the agent is aware of; and in such situations, the agents' inferences about harmful effects may be accurate.

The key insight in this paper then relates to addressing situations under the second law where an agent may disobey human orders due to its potential for causing harm to other humans: given the uncertainty described above, an agent should not arbitrarily disobey orders from humans, but must first address its own or the human users' uncertainty. It is only upon resolution of such uncertainty that the agent may then disobey an order if it causes harm to others.

The key technical innovation then is recognizing situations where an agent or the human user faces significant uncertainty, and taking actions to resolve such uncertainty. When addressing domains such as the office assistant, agents bring to bear POMDPs. Given this POMDP framework, when an agent is faced with an order from humans with the potential for harm (significant reduction in individual or team utility) it must consider two particular sources of uncertainty before obeying or disobeying such an order: (i) uncertainty about actual user progress on a task and (ii) uncertainty about user capability to perform a task(i.e.- user's rate of performing the task).

If the above two uncertainties are resolved, and the expected utility computations of the POMDP illustrate that the human order leads to reduction in individual or team utility, then this is the case (by Asimov's second law) where an agent may ultimately disobey human orders.

In addressing the simulated disaster rescue domain, the issue centers on potential uncertainty that a human user must face. Here, an agent team acting in the simulated disaster-rescue environment potentially has more certainty about the world state than the human user does. Unfortunately, the disaster is spreading

rapidly, and unlike the office environment, the agent team may have little time to deliberate on the uncertain situation, and come up with a detailed policy (as with a POMDP). Instead, the agent team quickly brings up to the notice of the human user key possible sources of potential harm due to the human's order. At this juncture, the human user may be able to reduce his/her uncertainty about the world state, and thus possibly rectify his/her order.

4 Operationalizing Asimov's Laws

In each of our two domains, appropriately obeying the first and second laws would have improved the situation. Specifically in the second law, the caveat where human directives should be followed, *unless it causes harm to humans*, is not being paid attention to. Instead, as mentioned in Section 2, agents blindly obey human commands, which is problematic. However, as mentioned earlier, in complex domains, there is significant uncertainty. Given such uncertainty, it is quite feasible for the humans to provide imperfect inputs or orders; yet agents may not be certain that these orders are in error, due to the uncertainty that they face. Our position is that in order to start constructing teams of agents and humans that perform well, they must not always take human input as final, yet must only do so after resolving uncertainty.

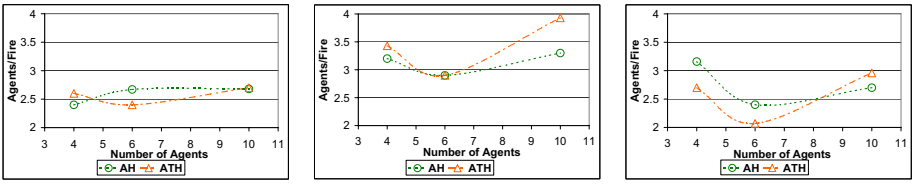
Given that humans may (unintentionally) provide problematic input, we propose that there are 5 general categories of agents' reactions to problematic human input. In particular, agents may:

- A. Follow the human input exactly
- B. Ignore the human input completely
- C. Make the human aware of the alleged problem in the input
- D. Make the human aware of the alleged problem in the input and offer non-problematic option(s)
- E. Limit human input to only pre-defined non-problematic options to be chosen from by the human

Due to the uncertainty (mentioned in Section 3.3), Option A and B become infeasible. Option A results in suboptimal performance and does not take advantage of the team's resources and potential as seen in Section 2. Option B may end up in better performance, but not only results in angry or confused humans, the agents may also be mistaken due to its own uncertainty and poor performance. Option E is not very practical (too many options to explore) for dynamic domains, or worse, it is impossible to elicit all options. It is then desirable to engage in some of the dialogue described in Options C or D. Our aim is to have joint performance of the agents and humans be better than either of them separately, that is to have the agents correct problems in humans and vice versa.

4.1 Disaster Response

Our goal was to have the agent team be able to detect the problematic input seen in the previous experiments and then be able to engage in some type of



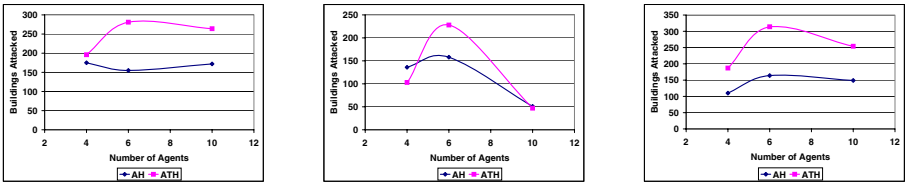
(a) Subject 1 (b) Subject 2 (c) Subject 3

Fig. 3. Amount of agents assigned per fire

dialogue with the human user. In order to do this, we continued an in depth analysis of what exactly was causing the degrading performance when 6 agents were at the disposal of the human user. Figure 3 shows the number agents on the x-axis and the average amount of fire engines allocated to each fire on the y-axis. AH and A_TH for 6 agents result in significantly less average fire engines per task (fire) and therefore lower average. For example, as seen in Figure 3, for the A_TH strategy, subject 3 averaged 2.7 agents assigned to each fire when 4 agents were available, whereas roughly 2.0 agents were assigned to each fire when 6 agents were available. It seems counterintuitive that when given more agents, the average amount that were assigned to each fire actually went down. Another interesting thing that we found was that this lower average was not due to the fact that the human user was overwhelmed and making less decisions (allocations). Figures 4(a), 4(b), and 4(c) all show how the number of buildings attacked do not go down in the case of 6 agents, where poor performance is seen.

We can conclude from this analysis that the degradation in performance occurred at 6 agents because fire engine teams were split up, leading to fewer fire-engines being allocated per building on average. Indeed, leaving fewer than 3 fire engines per fire leads to a significant reduction in fire extinguishing capability. Given this, we implemented the ability for the agent team to detect if a reallocation is pulling a teammate from a working group of 3 or more. Once this is detected, there is a high probability that the team performance will be degraded by following the human input. But since there is some uncertainty in the final outcome, the agents do not blindly follow (Option A from above) or ignore (Option B from above). Instead they present the possible problem to the human (Option C from above).

In order to evaluate this implementation we set up some initial experiments to determine the possible benefits of the team being able to reject the splitting of a coordinated subgroup. We used a team comprised of a human user and 6 agents. We used the same map and the same A_TH strategy as were used in previous experiments. Each of these results were from a short (50 time step) run of the DEFACTO system. The only variable was whether we allowed the agents to raise objections when given allocation orders to split up. In these experiments, the human user listened to all agent objections and did not override them. The results of this initial experiment can be seen in Table 1. In Table 1, we present



(a) Subject 1 (b) Subject 2 (c) Subject 3

Fig. 4. Number of buildings attacked

Table 1. Benefits to team when rejecting orders allows split of team. In top half, team accepted all human orders, and in bottom half, problematic orders were rejected.

Reject Orders to Split?	Buildings Damaged	Fires Extinguished
No	27	3
No	29	3
No	33	1
Yes	14	5
Yes	18	5
Yes	20	5

results for three problem instances. Performance is measured by calculating the amount of buildings damaged (less is better) and the number of fires extinguished (more is better). As seen from the number of buildings damaged, by allowing the agents to reject some of the human input (see bottom half of Table 1), they were able to more easily contain the fire and not allow it to spread to more buildings.

4.2 Office Assistants

In the TMP domain, the state space of the POMDP consists of a triple of three variables: {Progress on task, Capability level associated with the user, Time till deadline}. As mentioned earlier, of these three variables, an agent may be uncertain about the progress level and the capability level. Actions for the agent can be $\{Wait, Reallocate, AskDecision, AskProgress\}$. $Wait$ is for the agent to do waiting while user makes progress on the task, while *Reallocate* is for the agent to reallocate the task to a different user. *AskDecision* causes the agent to ask the user whether to reallocate, while *AskProgress* is for removing uncertainty about progress by gathering more information. *AskDecision* also leads to reduction of uncertainty in a user's capability level – an agent invokes *AskDecision* when, according to its estimate of user capability, a task should be reallocated. A user's agreement or disagreement to reallocate provides an opportunity to correct this estimate. However, an agent will disobey this order from a user at a later time if this estimate is corrected.

When an agent executes *AskDecision* to ask whether to reallocate, then, based on the user response, an agent addresses its uncertainty in user capability level as follows. There are three cases of user response to consider:

1. Reallocate: In this instance, the agent had concluded based on its estimate of the user's capability to reallocate. Since the user concurs, the agent's POMDP policy dictates that the task be reallocated to the appropriate user, and the policy terminates.
2. Don't Reallocate: Here the user has disagreed with the agent. Assuming a maximum error of ϵ in its estimation of the user's capability level, this response from the user makes the agent use a capability level increased by ϵ for the future time points. Consequently, the agent resolves its uncertainty in user capability level. If, at a later time, even with this increased user capability, the agent estimates that the user will be unable to fulfill the task, it will then reallocate, thereby overriding prior human input.
3. No Response: Equivalent to a wait action, with cost incurred for asking the user.

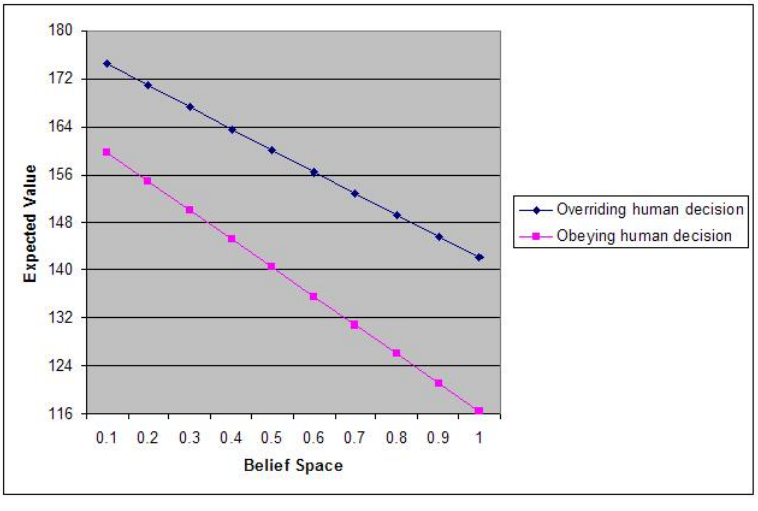


Fig. 5. Comparison of expected values of the two strategies

Figure 5 compares the expected value of two policies for the TMP problem mentioned in Section 2.2. The first policy is one where agents always obeyed human decisions, e.g. if the human gave an order that a task should not be reallocated, the agent absolutely never reallocated the task. In the second policy, the agents sometimes override human orders. In particular, if the human user ordered to *not reallocate*, then the agent initially obeyed the human order. While obeying this order, the agent also increased its estimate of human capability by maximum allowable amount (since the user disagreed with the agent's estimate of

reallocation), thus reducing potential uncertainty about human user capability. However, subsequently, the policy reallocated the task if the user was seen to be unable to finish the task on time, even though earlier the user had given an order to not reallocate. This is because the agent had now reduced its uncertainty about human capability, and was now certain that disobeying this user order will avoid harm to the team. In Figure 5, the y-axis plots the value of the two policies mentioned above: obeying human decision vs overriding human decision. The x-axis plots different belief states (probability distribution over world states). We see that the policy to (sometimes) override achieves higher expected value than the policy to always obey human decisions.

5 Related Work and Conclusion

Many projects that deal with agent interactions with humans have started to worry about the safety of those humans. Consequently, they have started to look to Asimov's laws of robotics for some crucial guarantees. This past work ([4,8,14]) has focused on asimovian agents but dealt with only the first law (against human harm). We have discussed the relationship of our work to this previous work extensively in Section 3. Another area of related work is mixed initiative planning [1,7]. For the most part, this work focuses on single-agent to single-human interactions, whereas we focus on multiagent teams. Additionally, our work is to allow for human-agent interaction during execution, as opposed to their work, which is focused on offline planning. None of this research addresses the issue of uncertainty addressed in our work.

Lastly, there has also been work where humans are beginning to interact with agent/robot teams [3,10]. These efforts recognize that humans may not provide a timely response. In part to alleviate the lack of such timely response, Scerri et al [9] introduced the notion of *adjustable autonomy strategies*. Our work already incorporates such strategies, but recognizes that human users may still provide such imperfect or incorrect inputs. There has also been some work that involves humans interacting with multiagent teams actually leading to performance degradation due to imperfect input: In past work [12] illustrated that an autonomous team of simulated robots performed better than when aided with human inputs (although in some situations the humans were able to improve the simulated robot performance). However, this work did not address the question of how the simulated robots would recover from such setbacks.

In conclusion, this paper is based on the premise that Asimov's laws provide us desirable guarantees for environments where humans must work with multiagent teams. While previous work has focused operationalizing just the first of Asimov's laws, this paper focused on the second law. In particular, the paper focused on the key insight provided by that law: agents must not blindly obey human inputs at all points. Instead, agents must pay attention to the caveat that in the law that allows for disobeying human inputs when such input leads to harm. Furthermore, we illustrated that given the uncertainty faced by the agent team and the human users, agents must attempt to reduce such uncertainty

before disobeying any human input. We illustrated the results of two different domains that each have different approaches to operationalizing Asimov's laws. In the disaster rescue simulation domain, real-time response precludes detailed planning to address uncertainty; whereas in our office assistant domains, agents performed detailed decision-theoretic planning to address uncertainty. These results show that the new agent-human teams avoid the original set of problems and provide useful behavioral guarantees.

References

1. James F. Allen, Lenhart K. Schubert, George Ferguson, Peter Heeman, Chung H Hwang, Tsuneaki Kato, Marc Light, Nathaniel G. Martin, Bradford W. Miller, Massimo Poesio, and David R. Traum. The trains project: A case study in defining a conversational planning agent. Technical report, Rochester, NY, USA, 1994.
2. Isaac Asimov. *Robot Visions (collection of robot stories).* Byron Preiss Visual Publications Inc, 1990.
3. Jacob W. Crandall, Curtis W. Nielsen, and Michael A. Goodrich. Towards predicting robot team performance. In *SMC*, 2003.
4. Diana F. Gordon. Asimovian adaptive agents. *JAIR*, 13:95–153, 2000.
5. http://www.ai.sri.com/project/CALO, http://calo.sri.com. *CALO: Cognitive Agent that Learns and Organizes*, 2003.
6. M. L. Littman L. P. Kaelbling and A. R. Cassandra. Planning and acting in partially observable stochastic domains. *AI Journal*, 1998.
7. K. Myers. Advisable planning systems. In *Advanced Planning Technology*, 1996.
8. D V. Pynadath and Milind Tambe. Revisiting asimov's first law: A response to the call to arms. In *Intelligent Agents VIII Proceedings of the International workshop on Agents, theories, architectures and languages (ATAL'01)*, 2001.
9. P. Scerri, D. Pynadath, and M. Tambe. Towards adjustable autonomy for the real world. *Journal of Artificial Intelligence Research*, 17:171–228, 2002.
10. P. Scerri, D. V. Pynadath, L. Johnson, P. Rosenbloom, N. Schurr, M. Si, and M. Tambe. A prototype infrastructure for distributed robot-agent-person teams. In *AAMAS*, 2003.
11. Nathan Schurr, Janusz Marecki, Paul Scerri, J. P. Lewis, and Milind Tambe. The defacto system: Training tool for incident commanders. In *The Seventeenth Innovative Applications of Artificial Intelligence Conference (IAAI)*, 2005.
12. Nathan Schurr, Paul Scerri, and Milind Tambe. Impact of human advice on agent teams: A preliminary report. In *Workshop on Humans and Multi-Agent Systems at AAMAS. 2003.*
13. P. Varakantham, R. Maheswaran, and M. Tambe. Exploiting belief bounds: Practical pomdps for personal assistant agents. In *AAMAS*, 2005.
14. D. Weld and O. Etzioni. The first law of robotics: A call to arms. In *AAAI*, Seattle, Washington, 1994. AAAI Press.

Persistent Architecture for Context Aware Lightweight Multi-agent System

Aqsa Bajwa[1], Sana Farooq[1], Obaid Malik[1], Sana Khalique[1], Hiroki Suguri[2], Hafiz Farooq Ahmad[2], and Arshad Ali[1]

[1] NUST Institute of Information Technology,
166-A, Street 9, Chaklala Scheme 3, Rawalpindi, Pakistan
Tel.: +92-51-9280658; Fax: +92-51-9280782
{aqsa.bajwa,sana.farooq,obaid.malik,sana.khalique,
arshad.ali}@niit.edu.pk
[2] Communication Technologies,
2-15-28 OmachiAoba-ku, Sendai 980-0804 Japan
Tel.: +81-22-222-2591; Fax: +81-22-222-2545
{suguri,farooq}@comtec.co.jp

Abstract. Application development on handheld devices using software agent technology is becoming more and more popular around the world. Escalation in the use of lightweight devices and PDA's leads us to create a concrete base for future nomadic applications, positioned in a changing environment. However, constrained characteristics of handheld devices serve as the main hindrance towards achieving this goal. This paper presents the architecture of context aware FIPA complaint multi agent system for the lightweight devices called SAGE-Lite, which are capable of providing fault tolerance through the mechanism of object persistence. Agents existing on lightweight devices can communicate and provide services via Bluetooth and the communication with the server is done via WAP. Since agents communicate via sending and receiving ACL messages, this architecture will minimize communication latency with in the platform. This framework allows implementing agent-based applications like business applications or e-commerce applications on these resource-constrained devices.

1 Introduction

Designing and implementing complex software systems have always been much complicated and time consuming. Different research communities have been striving to achieve better methods and techniques for the development of such systems and this struggle led to the basis of software agent technology, which a few years back became an active research area not only for academia but commercially as well. The research on wireless communication technology has also become popular lately as nomadic computing; wireless data communications and mobile devices enable accessing fixed network services from almost anywhere and any time [1].

R.H. Bordini et al. (Eds.): ProMAS 2006, LNAI 4411, pp. 57–69, 2007.

Software agent technology is a capable approach for the analysis, specification and implementation of complex software systems. Multi-agent systems are systems composed of multiple agents, which interact with one another, typically by exchanging messages through some computer network infrastructure. MAS provide proper execution environment to agents so that they can assure the provision of services to other agents by cooperating, coordinating, and negotiating.

It is about time that these two interesting and important technologies, nomadic computing and software agent technology, start to converge [1]. About 1/3 of the world's total population is mobile Internet users as handheld device are becoming more than a personal information system. So the communicative agent technology in our opinion will reach new heights in this direction.

Foundation for Intelligent Physical Agents or simply FIPA is one of the standard governing bodies [3], which provide an abstract architecture for lightweight multi-agent system developers to follow. Agent Management System (AMS), Agent Communication Language (ACL), and Message Transport Service (MTS) are one of its mandatory components. Figure 1 shows the list of mandatory and optional of lightweight MAS.

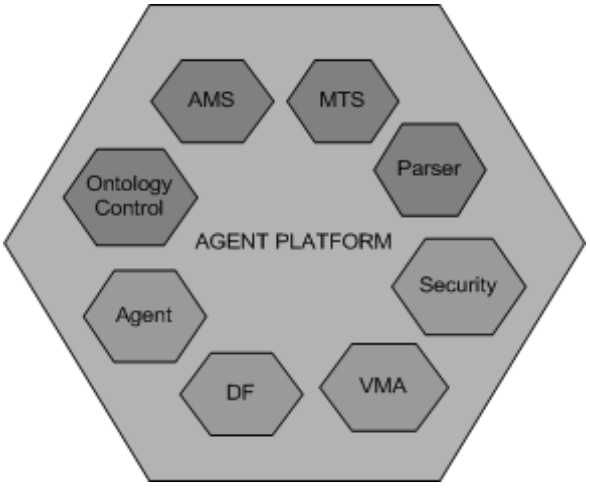


Fig. 1. The mandatory and optional components of a lightweight agent platform

Wireless devices have gained a lot of significance in the past few years. Diversified ranges of mobile applications are becoming popular day by day. The applications run using the constrained resources provided by today's vast variety of the cellular devices.

The resources in a handheld device are constrained in terms processing power, memory, graphical user interface support, screen size and permanent storage. On the network side, wireless networks are constrained by less bandwidth, more latency, less connection stability and less predictable availability.

At NUST-COMTEC we are developing a FIPA compliant lightweight multi-agent system called Scalable Fault Tolerant Agent Grooming Environment - Lite (SAGE-Lite). In this paper, we present our concept and detailed architecture of the FIPA (Foundation for intelligent physical agents) compliant lightweight multi-agent system. The primary focus of this paper is to give a complete grooming atmosphere to software agents in the limited environment of hand held devices where they can get registered and publish their services for other agents to discover and use. Software agents can also communicate with their peers on the same machine or on the remote machine via message transport service, which supports communication through WAP and Bluetooth.

This paper consists of 8 sections. Next section briefly reviews the related work. Section 3 describes the approach followed for this project. Section 4 gives the architecture of SAGE. Section 5 contains the SAGE-Lite architecture. Section 6 gives the difference between the lightweight architecture and non-lightweight architecture. Section 7 contains the detailed architectural features of SAGE-Lite. Programming details are in Section 8. Conclusion and discussion is done in Section 9.

2 Related Work

MAS are used as core technologies in various applications from information retrieval to business process automation. Most of MAS platform are based on JAVA runs on desktop or servers using J2SE and lack interoperability. SAGE-Lite can be compared with these lightweight agent platforms as they are not context-aware systems.

LEAP is a Lightweight Extensible Agent Platform; it enables the components of MAS to run on devices from PDA's to Smart Phones using either Java 2 Micro Edition (J2ME) or Personal Java, to desktops and servers running J2SE. It does not have support for object persistence to ensure fault tolerance. In case of failure in main system whole system will break down. Also LEAP does not provide efficient ACL message encoding implementation for wireless networks. FIPA specifies bit-efficient encoding scheme to be used to encode ACL message in wireless environment. Support for context awareness is not provided in LEAP as a result JADE agents cannot run on a variety of devices.

Micro FIPA-OS is an agent development toolkit and platform based on the FIPA-OS agent toolkit. The system targets PDA devices that have sufficient resources to execute a PersonalJava compatible virtual machine. The dependence on personalJava limits its deployment to relatively powerful PDA's. MicroFIPA-OS allows the use of FIPA-OS components such as AMS and DF. FIPA-OS and MicroFIPA-OS use tasks and conversations as the basic metaphor for programming agents, and agent functionality. Since the task and conversation management introduce overhead and latencies in messaging and agent execution. The platform is executed entirely on the device, but it is recommended that only one agent be executed on each small device.

AgentLite is an agent platform for deployment on fixed and mobile devices with various operating systems and that can operate over both fixed and wireless networks. It consists of an agent container running on each device on which several agents can be placed. If two agents on same device communicate with each other the agent container does the communication internally but if the agents are on separate devices then agent container act as a proxy. The architecture is not entirely FIPA complaint because there is no directory service. It can run on devices with J2ME/CLDC/MIDP. The smallest device targeted is mobile phone, although there are not any tests to prove that mobile phone can support a platform. [7].

Grasshopper is based on the older MASIF (Mobile Agent System Interoperability Facility) specifications of OMG (Object Management Group). It was intended for J2SE but a Personal Java version is also available. It concentrates on mobile agents and a stable extendible platform. Main drawbacks are lower scalability and a weak internal agent communication organization.[9]

3 Approach

Major design goals for SAGE-Lite were: to have a smallest footprint so that it can be deployed on devices with lowest memory and processing power. SAGE-Lite agents uses the ACL, a FIPA defined programming language for communication. Java 2 Standard Edition (J2SE) is not suitable for mobile devices; it requires too many resources and many standard classes are less relevant on small devices. PersonalJava is based on J2SE's Java Development Kit 1.1 and consist of Java Application Programming Interface (API) tailored for PDAs, set-top boxes, hi-end mobile phones. Unfortunately there is no 'one-size-fits -all' environment for a whole range of mobile devices with different capacities. Therefore, the development environment chosen for this project is Java 2 Micro Edition (J2ME), in which building blocks can be combined to setup a complete runtime environment that suits a particular type of device.

The requirements for the development of SAGE-Lite are:

- can be deployed on devices having at least 1Mb RAM.
- developed in Java 2 Micro Edition (J2ME). It is the reduced version of Java programming platform. J2ME technology consists of a Java Virtual Machine (JVM) and set of APIs for providing a complete runtime environment for the target device.
- target devices should support J2ME (CLDC 1.0 and MIDP 2.)

4 SAGE Architecture

Scalable fault tolerant agent grooming environment (SAGE) is the first research initiative of its kind in the South Asian region. SAGE is an open source FIPA complaint second-generation multi agent system, which is a distributed decentralized architecture.

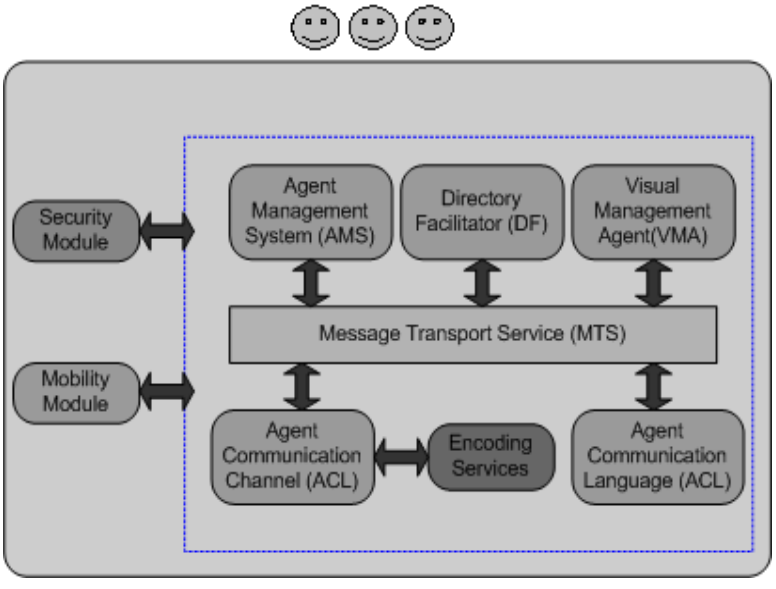


Fig. 2. Main Architecture of SAGE

This decentralization feature enables the system to be highly scalable and the objective of fault tolerance is achieved by the indigenous idea of Virtual Agent Cluster, which uses separate communication layers among different machines. The Virtual Agent Cluster works independently regardless of the external environment proceedings, providing a self-healing, proactive abstraction on top of all instances of multi-agent systems. Also the architecture fully supports peer-to-peer communication, which brings scalability, fault tolerance and load balancing among distributed peers as well [2].

5 SAGE-Lite Architecture

SAGE-Lite is an evolution of the SAGE agent platform from which it inherits the lightweight behavior. We propose surrogate architecture, a set-up in which SAGE and SAGE-Lite works together in combination. As shown in the fig. 3 there will be a static distributed platform SAGE consisting of main container providing FIPA interoperability to one or more lightweight platforms, SAGE-Lite. SAGE and SAGE-Lite will communicate using Wireless Application Protocol (WAP).

Agents running on different handheld devices will communication with one another via Bluetooth and WAP

Context Awareness. Lightweight devices have different features and capabilities. To cater the wide variety of devices context awareness module has been

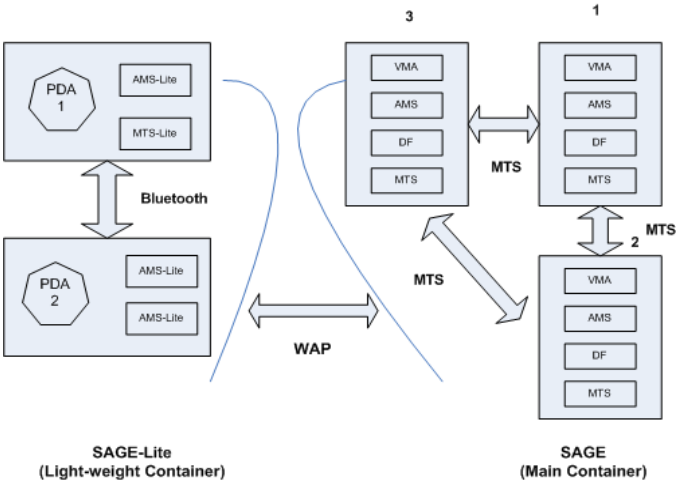


Fig. 3. Communication Between SAGE and SAGE-Lite

introduced in architecture. It enhances the capabilities of SAGE and makes it compatible with a vast variety of currently available lightweight devices. The concept presented in this architecture emphasizes on the device based context awareness i.e. the services should be provided to a devices depending upon the capabilities of the devices. If the device does not support GUI, then it should not get the service with GUI or if it has lower processing power then the services with minimal features should be sent to the devices. For this purpose the device will first send its full specifications to the main container i.e. SAGE and then SAGE will send the service to the device according to its specifications. Fig. 4 explains the above concept.

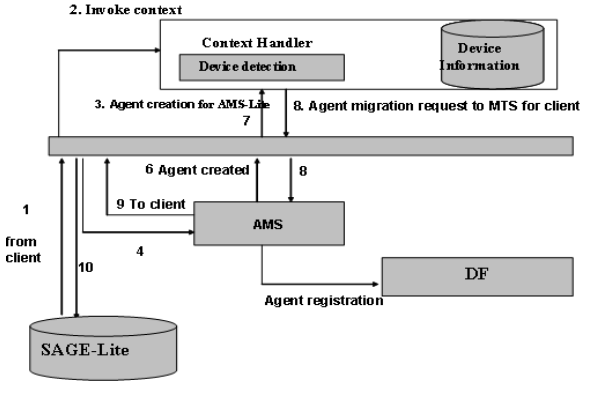


Fig. 4. Context awareness in SAGE

6 Differences Between SAGE and SAGE-Lite

Table 1. Difference between SAGE and SAGE-Lite

Name Of Component	**Non-Light Weight (SAGE)**	**Light weight (SAGE-Lite)**
AMS	-Heavy data structures. Each agent has its own record queue -Running as an Agent -All communication between components AMS, MTS, DF has to be done via ACL messages which incorporates encoding and decoding over heads with additional increase in execution memory footprint of platform. -Agent Migration is done using CORBA, which itself is heavy.	-One queue that maintains all agents record. No heavy data structures. -AMS acts as a local service -Communication between components AMS, DF, MTS does not need to be encoded using ACL thus reducing encoding/decoding, memory and processing overhead. -Agent migration is done by transferring the agent object via Bluetooth or WAP no additional packages e.g. CORBA are needed.
VMA	-Running as an agent -Heavy Graphical User Interface	-Small part of AMS not an agent -Limited GUI -Only available if the device has support for GUI -Availability and non- availability decided by Context Handler Module according to the device Specification sent by the device.
MTS	-Very heavy data structures -Multiple transmission and reception queues for a single agent that means 100's of queues for platform having many agents -Support for many protocols e.g IIOP, HTTP etc	-No heavy data structures -Only two queues one for transmission and one for reception handling all agents within the platform or from another platform. -Support for WAP and Bluetooth only
DF	-Built on HSQL including other database Packages. -Very Heavy, runs as a separate Agent (ACL encoding overhead)	-Is built on RMS, built in feature of J2ME so no additional Packages for database Management Required. -Runs as a service (local) and is incorporated as a small part of AMS
ACL	-Support for sl0, sl1, sl2, OWL etc -Full Support For Ontology	-Supports only sl0 -Minimal Support for Ontology

7 Features of SAGE-Lite

Our basic premise was to move from wired to wireless architecture. Existing lightweight architectures contain different system agents, created during the boot up time. These agents are started as separate threads and the code of these agents is already resident in the agent platform class files. Apart from these system agent there are application agents running on the handheld device too. All the communication that is being done among these agents is via encoded ACL Messages. Shortcoming of these architectures is that they consume a lot of processing power during encoding and decoding these ACL messages. SAGE-Lite on the other hand, instead of keeping AMS-Lite, DF-Lite and VMS-Lite as system agents, keeps them as system services. As a result there is a significant decrease in the consumption of processing power in SAGE-Lite. These system services call functions of each

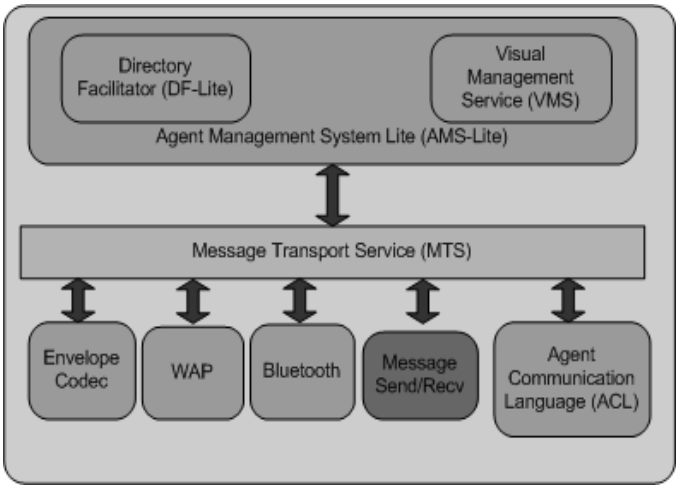


Fig. 5. Main system architecture of SAGE-Lite

other's to communicate. Hence there are no system agents on the platform but only application agents using all the resources to the maximum.

Major Components. The major components of SAGE-Lite are, AMS-Lite (Light-weight Agent Management System) responsible for managing the entire lightweight platform. It has DF-Lite (Light-weight Directory Facilitator) and VMS-Lite (Light-weight Visual Management Service) as its subcomponents, where DF-Lite serves as the yellow page directory on the hand held device and VMS-Lite provides the graphical user interface. MTS-Lite (light weight message transport service) is the hot communication link between the agents and ACL Lite (light weight agent communication language) is the language that agents use. The complete architecture of SAGE-Lite is given in the fig. 5.

Architecture of AMS-Lite. According to FIPA, AMS is the mandatory component of MAS. Only one AMS can exist in a single agent platform (AP). It maintains a directory of AIDs, which contain transport addresses (amongst other things) for agents registered with the AP. Each agent must register with an AMS in order to get a valid AID. In SAGE-Lite, AMS-Lite contains DF-Lite and VMS-Lite as its sub components.

Directory Facilitator Lite (DF-Lite). When a request for agent creation is received, AMS-Lite generates valid AID and the agent details are saved in DF-Lite. DF-Lite provides yellow page services to other agents. Agents can also register their services with it. To find out what services are offered by other agents they need to query AMS-Lite, which in turn queries the DF-Lite.

Rationale of having a local DF (DF-Lite) is that a mobile agent is not bound to a device or a platform, which created it. It moves from one platform to another

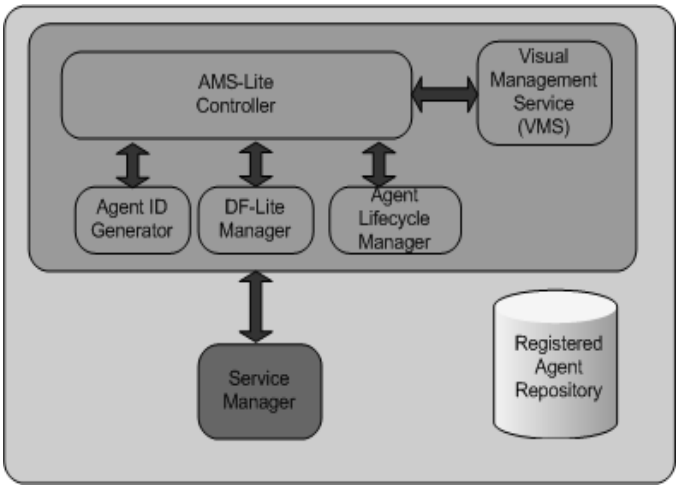


Fig. 6. Architecture of Lightweight Agent Management System

or to one device to another to complete its task.. This local DF-Lite on a light device contains the lists of agents running only on that particular device. In this way each device will have a list of its all registered agents in its local DF-Lite, and all the registered agents on the whole platform will be listed in the DF of SAGE which is our main container. This meticulous feature makes SAGE-Lite highly distributed or distributed mobile container. This feature mainly helps when a device shifts itself from one cell to another. Portable devices roam about in different location in an unpredictable way. And they have no prior knowledge about the available services and resources; they need to discover the available services on run time. So when our mobile device enters a new zone, record of all the agents residing on the device will also be shifted into a new zone through this DF-Lite. Otherwise our device needs to remain connected with home agent platform (main container residing in the previous zone). Without DF-Lite, the only possible way for the device to get its agents registered with the guest server (main container residing in new zone) would be by letting the guest server contact the home server for the list of agents running on that device. For this purpose, the lightweight device after movement from one zone to the other would be required to send its name with the name of the home platform to the guest server. Alternatively, the device can remain connected to its home platform and in this case it does not have to bother the guest sever, but there will be a disadvantages with full time connectivity with the home server. The former technique has three major disadvantages:

– The wired link between different servers can suffer a severe bottleneck problem. The mobility factor of the handheld devices makes them shift there cells every now and then plus keeping in mind the increasing number of cell phone users, it is quite evident that more then reasonable amount of traffic can be

generated which can cause the links to some servers suffer from bottle neck problems.[8]
- The time the guest server takes to get the list of registered agents on the device causes delay in the normal functioning of the application running on SAGE-Lite platform. [8]
- Another reason is that when two devices will search for services using Bluetooth and there is no local DF, then each device needs to connect with its server every time request for services arises. [7]

Services published by agents on a lightweight device are discoverable by other agents residing on different lightweight devices through the Bluetooth by service discovery mechanism.

Architecture of MTS-Lite. While designing MTS-Lite, the major point of consideration was to keep this module as light as possible in terms of the memory footprint it requires. It will enable the module to operate efficiently in the constrained environment provided by the small devices. This concern limited us from the use of heavy data structures or larger class hierarchies through out the development process. Fig. 8 shows the proposed class hierarchy for MTS-Lite. Also sending ACL messages over a wireless link had its own complications and risks. To make this communication efficient and effective, we adopted a FIPA-complaint encoding scheme called "Bit Efficient Encoding Scheme" for the representing ACL Messages to be sent over the wireless link. Fig. 7 shows the isolated architecture of MTS-Lite. This module is capable of communicating wirelessly through WAP and Bluetooth. The Message Send/Receive Module is responsible for the message buffering and envelope codec is responsible for the encoding of the envelope in Bit Efficient encoding representation. Fig.7 shows the MTS-Lite Architecture.

In order to minimize the processing power and memory consumed by the message buffering, only two queues will be maintained, one for the received

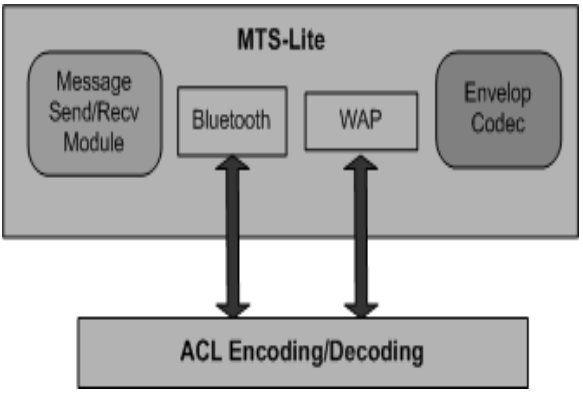


Fig. 7. MTS-Lite Architecture

messages and the other for the messages to be transmitted. The WAP and the Bluetooth clients will keep on reading the queue for a message and as soon as they get it, the message is transported after encoding. When MTS-Lite receives the message to be sent, it first of all checks its own DF-Lite to see if the agent resides on the same machine. If it finds the match on the same machine then instead of putting the message in the transmission queue, the message is directly put into the reception queue.

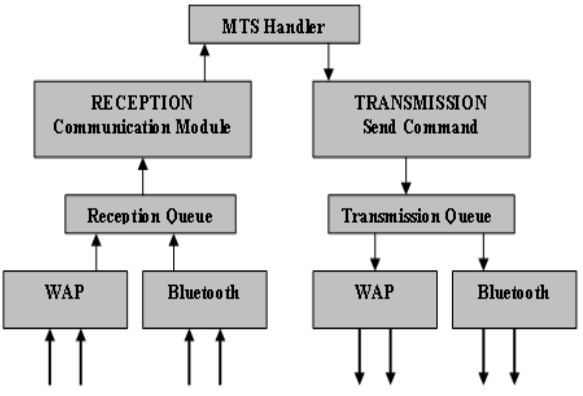


Fig. 8. Proposed class hierarchy for MTS-Lite

However if the message is to be sent to an agent executing on another machine, then the message is put into the transmission queue. For the message to be sent over to another platform, the envelope is also encoded using the particular codec.

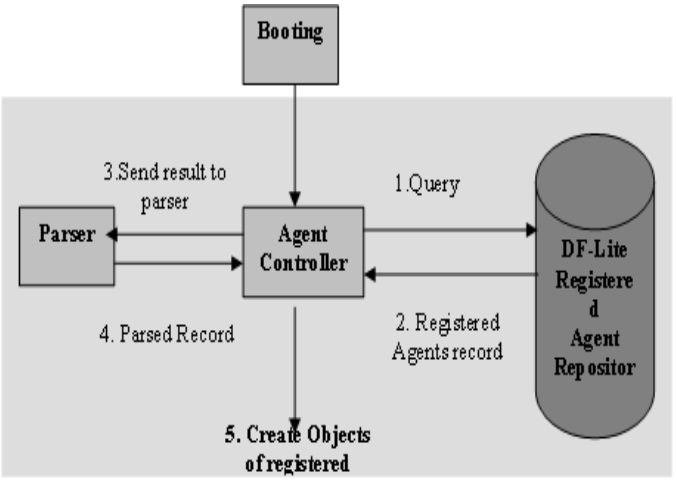


Fig. 9. Object persistence mechanism in SAGE Lite

Fault- tolerance through Object persistence. Agents record is stored in byte format using the Record Management Source (RMS), which is a built in feature of J2ME. For storing and manipulation of data stored, RMS API and Enumeration API have been used. SAGE-Lite provides fault tolerance in a system through object persistence. In existing systems when the platform crashes, data of all the registered agents is lost and when the system boots again, the agents need to be registered again with the platform's AMS. In SAGE Lite, when the platform crashes, provides recovery support by specifically maintaining a list in RMS of all the registered agents running at the time of crash and then brings them back into their respective states after the system boots up again. This makes the architecture persistent. Fig.9 explains the above-mentioned concept.

8 Programming

- Agents and their GUI separated using dynamic linking after the decision of the context handler at the server running SAGE.
- All agents extend from the ServiceAgent Class.
- The context handler sends the GUI of the agent separately to the device
- The linking of the GUI with the agent is done at the run time at the server.
- All GUI's of agents have a link function in which the reference to the agent for which the GUI is made is stored.
  public void link (ServiceAgent agent)
  AgentForGui=agent;
- Upon reception of the GUI at the client device (PDA, mobile) the agent controller check and verifies the agent and its GUI for link Authenticity

  Public boolean verifyGUILink(ServiceAgent agent,GUI gui)
  If(gui.AgentForGUI.name.equals(agent.name))
  If(gui.AgentForGui.agentId.equals(agent.agentId))
  If(gui.AgentForGui.Creator.equals(agent.Creator))
  return true;
  return false;

If there is no link error i.e. the source agent for the GUI and the Agent in check are same then the AgentController registers the agent and its GUI in the Local DF (DF-Lite).

9 Conclusion

We have developed a context aware lightweight multi agent system called SAGE-Lite. Framework allows implementing agent-based business applications or e-commerce applications on these resource-constrained devices. By developing prototypical applications based on the framework, we have shown that it may in fact be utilized to develop context aware services efficiently. Therefore a lot of emphasis has been given to not only on the development of an effective system

but also a system that is fully resource efficient. The proposed architecture is enriched with a lot of unique ideas, which have and will pave ways for a lot more research initiatives in times to come. Future work is also required to ensure that our agents can handle the unreliable nature of the network messages.

References

1. Supporting Nomadic Agent-based Applications in FIPA Agent Architecture - Heikki Helin.
2. Arshad Ali, Sarmad Malik, Muazzam Mugal, M. Omair Shafiq, Amina Tariq ,Amna Basharat, NUST Institute of Information Technology (NIIT) National University of Sciences and Technology,(NUST) ,Rawalpindi, Pakistan. Scalable fault tolerant Agent Grooming Environment - SAGE.
3. Foundation for Intelligent Physical Agents. http://www.fipa.org.
4. FIPA Nomadic Application Support Specification, Document No [SI00014H].
5. FIPA Agent Management Specification, Document No [SC00023K].
6. FIPA Message Transport Specification, Document No [0C00024D].
7. Cosmin Carabelea*, Olivier Boissier Ecole Nationale Supérieure des Mines de Saint-Etienne Centre SIMMO-SMA, 158 Cours Fauriel, 42000, St.Etienne, France “Multi-Agent Platforms on Smart Devices : Dream or Reality ?”.
8. Alf Inge Wang* Carl-Fredrik Sørensen, Eva Indal, Dept. of Computer and Information Science, Norwagian University of Science and Technology, N-7491 Trondheim, Norway. “A Mobile Agent Architecture for Heterogeneous devices”.
9. Bernaer, Stijn De Causmaecker, Patrick Maervoet, Joris Vanden Berghe, Greet “ An agent framework for effective data transfer”.

Architectural Design of Component-Based Agents: A Behavior-Based Approach

Jean-Pierre Briot[1,2], Thomas Meurisse[1], and Frédéric Peschanski[1]

[1] Laboratoire d'Informatique de Paris 6 (LIP6)
Université Paris 6 - CNRS
Case 169, 4 place Jussieu
75252 Paris Cedex 05, France
{Jean-Pierre.Briot,Thomas.Meurisse,Frederic.Peschanski}@lip6.fr
[2] Currently visiting CS Dept., PUC-Rio, Rio de Janeiro, Brazil

Abstract. This paper relates an experience in using a component model to design and construct agents. After discussing various rationales and architectural styles for decomposing an agent architecture, we describe a model of component for agents, named MALEVA. In this model, components encapsulate various units of agent behaviors (e.g., follow gradient, flee, reproduce). It provides an explicit notion of control flow between components (reified through specific control ports, connexions and components), for a fine grain control of activation and scheduling. Moreover, a notion of composite component allows complex behaviors to be constructed from simpler ones. Two examples, in the domain of multi-agent based simulation, are presented in this paper. They illustrate the ability of the model to facilitate both bottom-up and top-down approaches for agent design and construction and also to help at different types of potential reuse.

Keywords: component, agent, multi-agent systems, behavior, design, composition, architecture, simulation.

1 Introduction

Components and multi-agent systems are among current popular approaches for designing and constructing software. Both of them propose abstractions to organize software as a combination of software elements, with easier management of evolution (such as changing and adding elements). We consider that multi-agent systems push further the level of abstraction and the flexibility of component coupling, notably through self-organization abilities [5]. Meanwhile, we believe that the component concept and technology may help in the actual construction of multi-agent systems:

– at the *system level*, we may consider each agent as a component, to provide some support for integration, configuration, packaging and distribution of multi-agent systems,

R.H. Bordini et al. (Eds.): ProMAS 2006, LNAI 4411, pp. 71–90, 2007.

– at the *agent level*, by providing some support for structuration, (de)composition and reuse of its internal architecture.

In this paper, we focus on the second category. Indeed, we believe that the design and construction of an individual agent can benefit from the principles of software components (encapsulation, explicit connectors...). Our objective is to help in an incremental design of agents as the composition of simpler agent behaviors and activities (e.g., follow gradient, flee, reproduce...). Our main application field target is multi-agent-based simulation of phenomena (biological, ecological, social, economical...). Their specific requirements definitely influenced our design decisions, as well as our case studies and applications conducted. Meanwhile, we believe that the scope of the component model that we propose goes beyond the domain of multi-agent-based simulations, and that other application areas could benefit from some of its principles, e.g., making control available at the composition level.

After first discussing some rationales for the design of component-based agent architectures, and referring to related work, we describe a component model named MALEVA, which aims at encapsulating and composing units of behaviors to describe complex agent architectures. This component model does not impose a specific architectural style. One of its specificities is that it applies the principles of components and software composition to the specification of control, through the notions of control ports and control components. Two examples will be presented in the paper. They illustrate how MALEVA can support bottom up as well as top down design, and also how it offers some potential for reuse and specialization, through: structural composition of behaviors, abstract behaviors and design patterns.

2 Rationales and Styles for Agent Architectures

We consider an *agent architecture* as the description of the relations between the software (or sometimes hardware) modules that implement the various agent functions. Except for simple reactive agents, the architecture of an agent may be complex. It is thus useful to describe it in terms of simpler lower level components that interact with each other.

Inspired by the seminal work on software architectures by Shaw and Garlan [23], we tentatively propose a classification for agent architectures from the perspective of *architectural styles*.[1] In this paper, we focus on the rationales for decomposition and on their impact on the reuse of the architecture or/and of its components. It is important to note that we do not expect our typology to be exhaustive. Also note that, as for software architectures [23], a complex architecture (e.g., InteRRaP, see Section 2.3) may juxtapose and combine several architectural/decomposition styles.

[1] There is of course no unique typology for agent architectures, and we may find other classifications in the literature (e.g., in [20]), such as horizontal/vertical or reactive/cognitive/hybrid.

2.1 Cycle-Based Style

The architectural style based on the notion of cycle, among the simplest ones, follows the basic computational cycle of an agent situated within an environment: perception (of the environment), state update (data or/and mental state), generation of intentions (of actions), action. An example is a general architecture for situated reactive agents, introduced in Section 4.1. Another example is Yoav Shoham's Agent-Oriented Programming (AOP) architecture for cognitive agents [24] (see Figure 1).

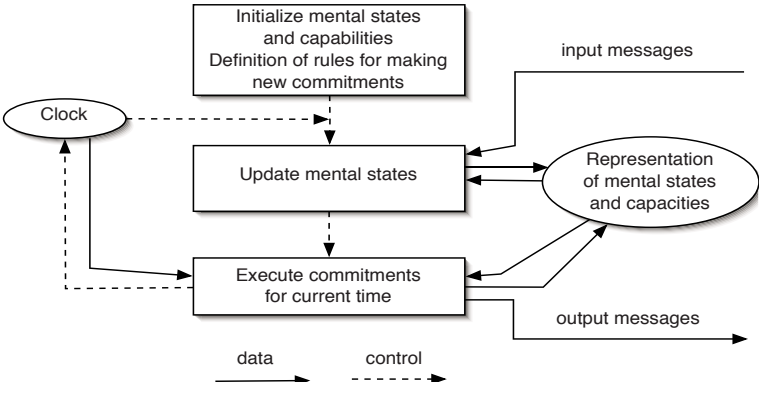


Fig. 1. AOP Architecture

2.2 View-Based Style

Another style of decomposition, more structural than computational, considers various view points (e.g., interaction, environment, organization...) and their respective units of processing (e.g., perception, communication, coordination...). An example is the VOLCANO architecture [21], which decomposes an agent along four dimensions: A (agent), E (environment), I (interaction) and O (organization). The architecture is actually a *framework* with components (named bricks) `A`, `E`, `I` and `O`. Figure 2, illustrates the central position of the `A` brick. Note that the designer needs also to implement inter-bricks adaptors/wrappers, respectively `AE`, `AI`, `AO`, `EI`, `EO` and `IO`.

Another example is the generic model of agent architecture (Generic Agent Model: GAM) [4], based on the DESIRE methodology and component model [3]. It includes a set of components (e.g., interaction management, information maintenance) each dedicated to a specific type of processing (see Figure 3, imported from [4]). The GAM generic model (also a framework) has been instantiated to model (retro-engineer) various agent architectures, such as BDI, and ARCHON.

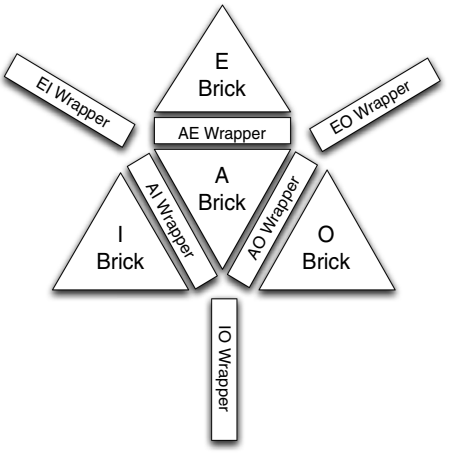


Fig. 2. VOLCANO architecture

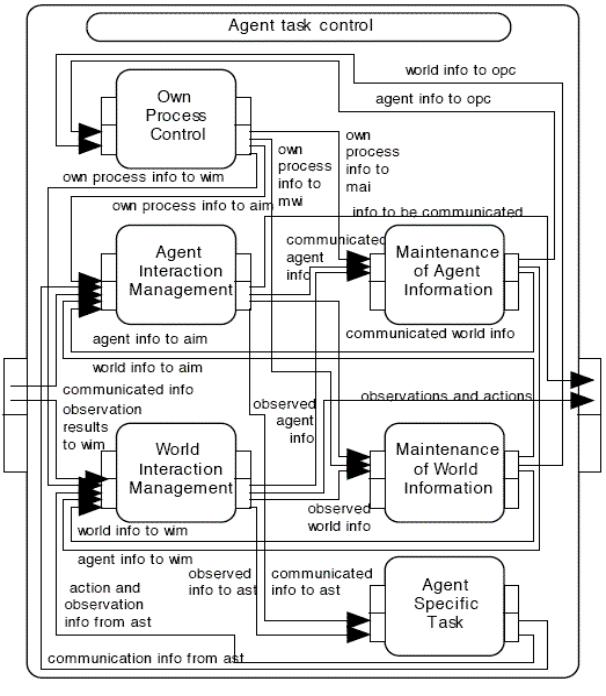


Fig. 3. DESIRE GAM architecture

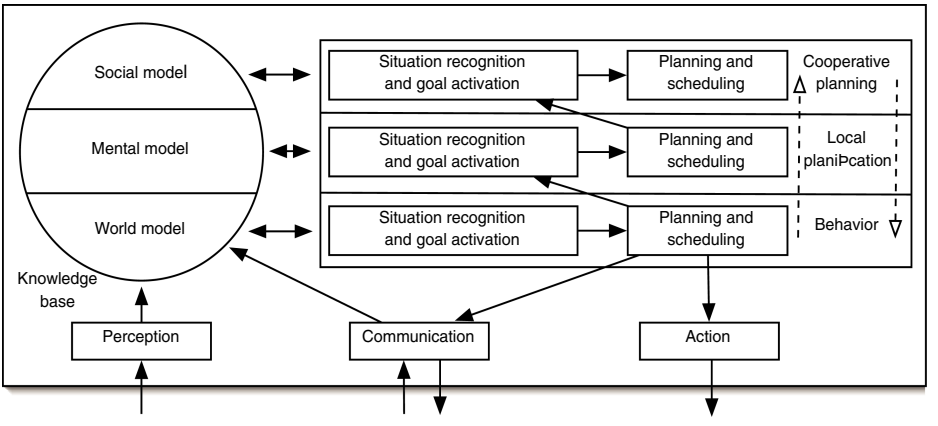


Fig. 4. InteRRaP architecture

2.3 Level-Based Style

Another approach considers different various levels (and models) of knowledge, reasoning and action, to structure the architecture, e.g., through the distinction between world model, self model, and social model. A representative example is the InteRRaP architecture [19] (see Figure 4, adapted from [19]). InterRRaP is structured as a hierarchy of three layers, concurrently active: cooperative planning, local planning, and reactive behaviour. The internal architecture of each level follows the same model, based on situation recognition and planning. A knowledge base structures the information manipulated by each layer, thus respectively: social model, mental model, and world model. Two dual control mechanisms between layers are considered: upwards activation request, to activate the layer above, and downwards commitment signal, to delegate execution of commitments to the layer below.

2.4 Behavior-Based Style

A more radical style of decomposition, considers basic behaviors of the agent as the units of (de)composition. An example is Rodney Brooks' subsumption architecture [7], in which various behaviors (e.g., random move, obstacle avoidance...) are simultaneously active. They are organized within some fixed hierarchy and their associated priorities (see Figure 5, adapted from [7]). In practice, a behavior may replace input data of the behavior situated below, as well as inhibit its output data (for instance, in case of close obstacle perception, the obstacle avoidance behavior may take control over other ones).

2.5 Discussion

The architectures that we surveyed are usually more tailored at a specific model of agent (e.g., cognitive collaborative agent for the InteRRaP architecture,

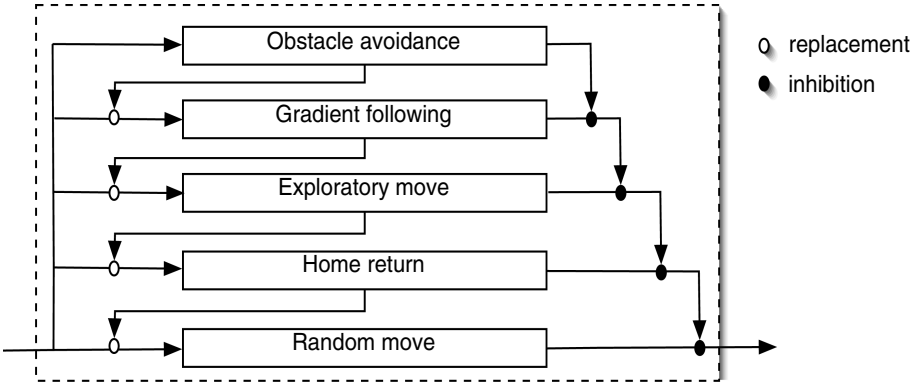


Fig. 5. Subsumption architecture

situated agent or robot for subsomption architecture). They aim at genericity, but, in practice, they may not provide enough flexibility. For instance, it is often uneasy, in some case almost impossible, to replace and add, and moreover to remove components. Also, the implementation of the architecture does not always follows the basic requirements of software components (output interfaces, explicit connectors...),[2] neither classical component models (e.g., JavaBeans). The VOLCANO architecture clearly separates the components, but in order to replace one component by another one, we are forced to re-implement the corresponding adaptors. The subsumption architecture actually represents an abstract model of architecture, instantiated for a specific robot and objective (e.g., see Figure 5). It is concise but also difficult to evolve (e.g., add a component), as the fixed hierarchy is the key of control between components.

We think that the behavior-based style of decomposition, as used in the subsumption architecture, is somewhat radical, but it is also the closest to the concept that we aim at decomposing: the *behavior* of the agent. A radical option is then to only offer a model of component, in a way similar to a general software component model such as JavaBeans, without a specific agent architecture. The main difference is that the control aspects, and not just the functionalities, must be also made composable in a flexible and open way, in order to replace the fixed hierarchical control model of the subsumption architecture. We propose that, by keeping in line with the idea of a component model, control is specified/reified through control ports and connexions (see Section 3.1), in order to represent arbitrary patterns of control flow.

[2] The JADE architecture [1] offers some basic support for the designer to construct an agent as a set of *behaviors* (instances of class `Behaviour`). Some subclasses, e.g., `CompositeBehaviour` and `ParallelBehaviour`, provide basic structures for constructing hierarchies of behaviors or/and for expressing control structures, e.g., the most advanced one, `FSMBehaviour`, relies on finite state automata. Meanwhile, JADE behaviors are not real components (no output interface/ports nor connectors), thus the architecture of an agent is still partly hidden within the code.

3 The MALEVA Model of Component

As has been explained above, the objective of the MALEVA agent component model is to help in incremental design and construction of agent behaviors by composing simpler behaviors, encapsulated as software components.

3.1 Data Flow and Control Flow

In MALEVA, a distinction is made between the activation control flow and the data flow connecting the components. As we show in Section 3.2, this characteristic and specificity of our model, which decouples the functional architecture from the activation control architecture, makes components more independent of their activation logic and thus more reusable. Consequently, we consider two different kinds of ports within a component:

- *data ports*. They are used to convey data transfer (one way) between components. Note that data ports are typed, as discussed in Section 6.1.
- *control ports*. A behavior encapsulated in a component is activated only when it explicitly receives an activation signal through its input control port. When the execution of the behavior is completed, the activation signal is transfered to its output control port.

In addition to the *semantic* distinction between data ports and control ports, specific to MALEVA, we find the common *structural* distinction between input ports and output ports. Table 1 summarizes that.

Table 1. Data and control ports

	Data	*Control*
Input port	Data consumption	Activation entry point
Output port	Data production	Activation exit point
Connexion	Data transfer	Activation transfer

3.2 An Introductory Example

Figure 6 shows a first and very simple example of assembly/composition of components: a sequence of two components. Component `B` is activated after the computation of component `A` completes. Regarding data, component `B` will consume the data produced by component `A` only after computation of `A` completes. In this figure, as well as the following ones, data flow connexions are shown in solid lines, and control flow connexions in dotted lines.

Figure 7 recombines the two same components, but this time activated concurrently. Note that the control connexions have been changed accordingly, but not the data connexions. The semantic is analog to the *pipes and filters* [23]

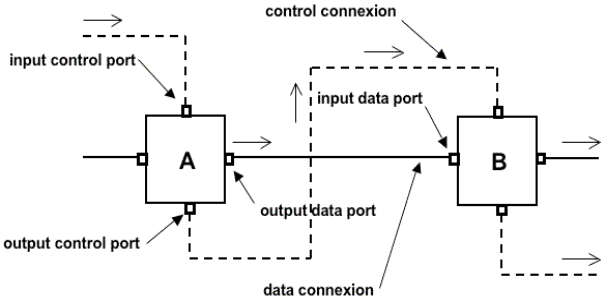


Fig. 6. Sequential activation of two components

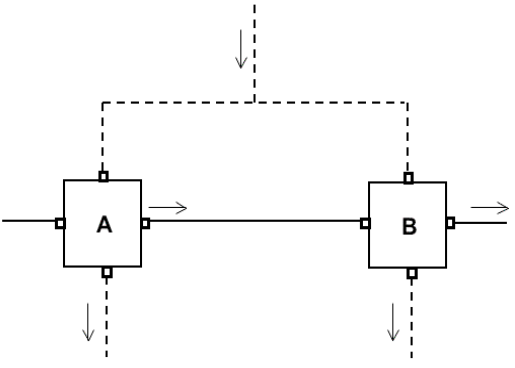


Fig. 7. Concurrent activation of two components

architectural style: component `B` consumes what `A` produces while they are both active simultaneously.

This simple example is a first illustration of the possibilities and flexibility in controlling activation of components. One may describe active autonomous components (with an associated thread), explicit sequencing or any other form of combination.[3] Flow of control is specified outside of the components, which provides more genericity on the use of components. The designer of the application also has a fine grained control over activation policies between components. Making possible the control of temporal dependences between behaviors - which are usually left implicit -, independently of behaviors functionalities, helps experts at experimenting with various strategies, at comparing results with the target models, and at quantifying the impact on biases [6].[4]

[3] An additional dimension, the mode of activation of components, which could be asynchronous or synchronous, will be briefly addressed in Section 6.2.

[4] For instance, [14] shows that results of simulations can be found biased in cases where the scheduling of the actions within an agent remains deterministic.

3.3 Designing Agent Behaviors

Designing and constructing agents for a given application should ideally consist mostly in assembling existing behavior components. We therefore assume that there is a library of behavior components associated to the application domains targeted. A component may be primitive (the behavior is written in the underlying language, e.g., Java) or *composite*,[5] as the encapsulation of a composition (assembly) of components.

4 A First Example: Bottom-Up Design of Prey and Predator

The MALEVA model has been particularly targeted at and used for multi-agent-based simulation (MABS) applications [22], in various domains such as ecology, ethology, and economy. In multi-agent-based simulations, various elements of the phenomena modeled and their interactions are explicitly modeled and studied. The two examples described in this paper show some facets of design and of potential reuse.

Our first example will define behaviors of situated agents within an ecosystem. First step is thus to define a general architecture for situated agents.[6]

4.1 Abstract Architecture of a Situated Agent

A situated agent senses its environment (e.g., position of the various agents near by, presence of obstacles, presence of pheromones...) through its sensors. These data are used by its (internal) behavior to produce data for its effectors, which will act upon the environment (e.g., move, take food, leave a pheromone, die...). The general architecture of a situated agent usually follows the computational cycle:

$$sensors \rightarrow behavior \rightarrow effectors$$

and is shown at Figure 8.

4.2 Prey Behavior

We will now define and construct the basic behaviors of preys and predators. By following a bottom up approach, we first define a set of elementary components,

[5] It corresponds to a notion of *structural composition* as opposed to, or rather *in addition to*, *functional composition* (simple assemblage). Such encapsulation of assemblages of components represents a very powerful abstraction principle. Of course, a composite may provide extra functionalities (and control specifications) at its higher abstraction level, making it a true component on its own. Another example supporting the notion of composite component is the Fractal component model [8].

[6] Note that for other applications, e.g., micro-simulation [6], agents are not necessarily situated (within an environment) and thus do not use any sensor/effector. Other applications agents could also use inter-agent communication (ACL) modules.

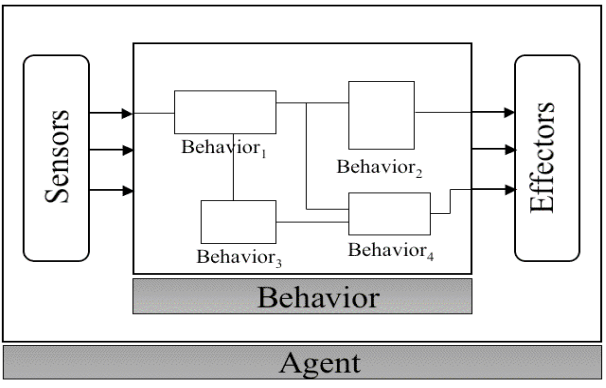


Fig. 8. General architecture of a situated agent

representing the basic behaviors of preys and predators, that we name: `Flee` (fleeing a predator), `Follow` (following a prey), and `Exploration` (exploration through a random move, which represents the default behavior). Then we will compose them, to represent the following agent behaviors: `Prey` and `Predator`.

A prey flees the predators being located within its field of perception. If no predator is close (sensed), the prey explores its surroundings by moving randomly. Thus, we construct the `Prey` behavior as the composition of the following three components: `Flee`, `Exploration`, and a control component named `Switch`.

4.3 Control Components

The `Switch` control component reifies the standard conditional structure into a special kind of primitive component.[7] The condition is the presence or absence of an input data. The behavior of `Switch`, once being activated (receiving an activation signal), is as follows:

IF data is received through `If` (input data port)
THEN transfer control through `Then` (output control port)
AND send data through `Then` (output data port)
ELSE transfer control through `Else` (output control port)

The architecture of the `Prey` behavior follows this pattern and is shown at Figure 9. If a predator has been detected (some data representing the predator loca-

[7] Note that the MALEVA standard library includes other control components, analog to standard control structures (e.g., repeat loop) or synchronization operators (e.g., barrier synchronization) [16]. They will not be described in this paper.

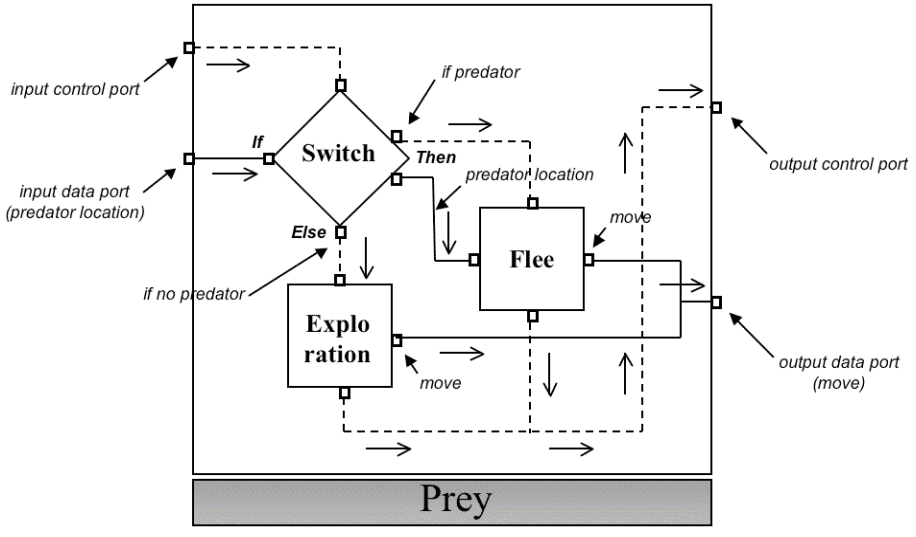


Fig. 9. `Prey` behavior

tion has been received on the input data port).[8] `Switch` transfers control through its `Then` output control port, which activates the `Flee` behavior. Then `Flee` can compute a move data based on the location of the predator, and send it through its output data port. The move data is finally transfered to `Prey` output data port and then to the effector, to produce a move of the agent on the environment. If no predator has been sensed (no data received), `Switch` transfers control through its `Else` output control port, which activates `Exploration` behavior. Note that `Exploration` does not need a data input to produce a move data.

4.4 Predator Behavior

We may now reuse the `Prey` behavior component to construct the behavior of a predator which follows the preys while fleeing his fellows predators, and otherwise explores its surroundings. The predator behavior may be defined as a prey behavior (it flees other predators and otherwise carries out an exploration movement), to which is added a behavior of predation (it follows preys that he could perceive). According to our compositional approach, we define `Predator` behavior component as a new composite behavior embedding *as it is* the existing `Prey` behavior component (see the result in Figure 10). Note that in our current design, hunger (predation) has priority over fear (fleeing), as `Prey` is activated by `Predator`. Other combinations could be possible.

[8] We assume that the input data port (perception of a predator in the environment) and the output data port of the `Prey` behavior have been connected to the corresponding sensor and effector data ports, along the general architecture of a situated agent, shown at Figure 8.

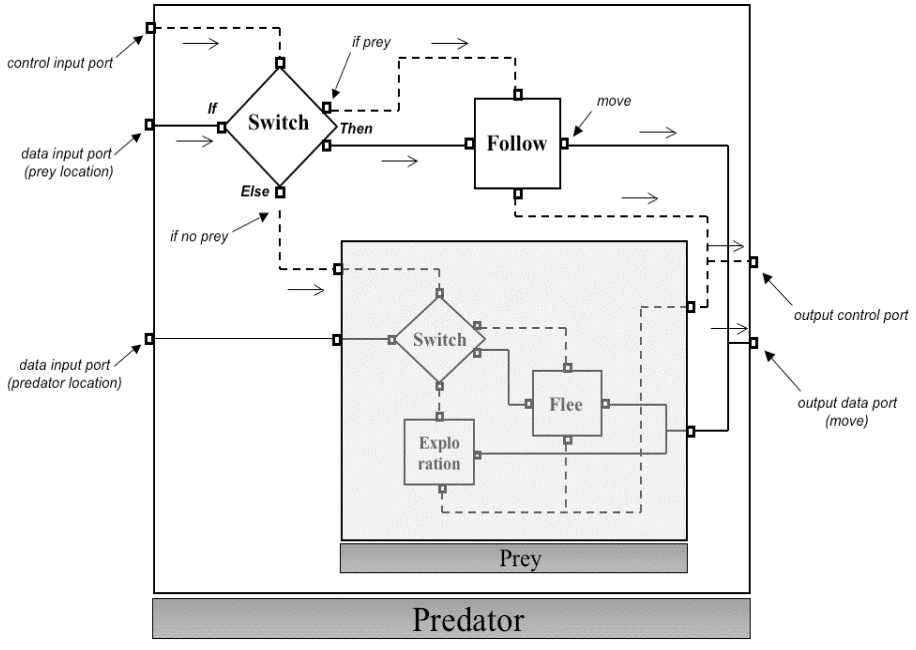


Fig. 10. `Predator` behavior (with `Prey` as a sub-component)

5 A Second Example: Top-Down Design of Ants

This second example illustrates a top down design of agent behaviors. The complete application was the reengineering in MALEVA [15] of the modeling and simulation of ant colonies for the study of their sociogenesis (Alexis Drogoul's MANTA framework [10]). Various types of ant agents are considered: eggs, larvae, worker ants, queens.[9] In this paper, we focus on the top down design of the behavior of an ant worker.

5.1 The Living Pattern

The first step of our design identifies some feature common to each living agent, the ability to age (and ultimately to die). Therefore, we design a behavior, partly abstract, named `Living`, shown at Figure 11. It includes 4 sub-behaviors/components: behavior `CheckAgeLimit` (it includes a variable `age`, incremented for each activation step and compared with the agent age limit); behavior `Die`; abstract behavior `Behavior`; and a `Switch` control component. When the agent reaches its age limit, `CheckAgeLimit` emits a `die` data. Then `Switch` activates `Die`, which in turn emits `suicide` data, ultimately conveyed to the actuators

[9] The metamorphosis process - from egg to larva and then to ant or queen - leads to the issue of behavior evolution and architectural dynamicity, see Section 8.

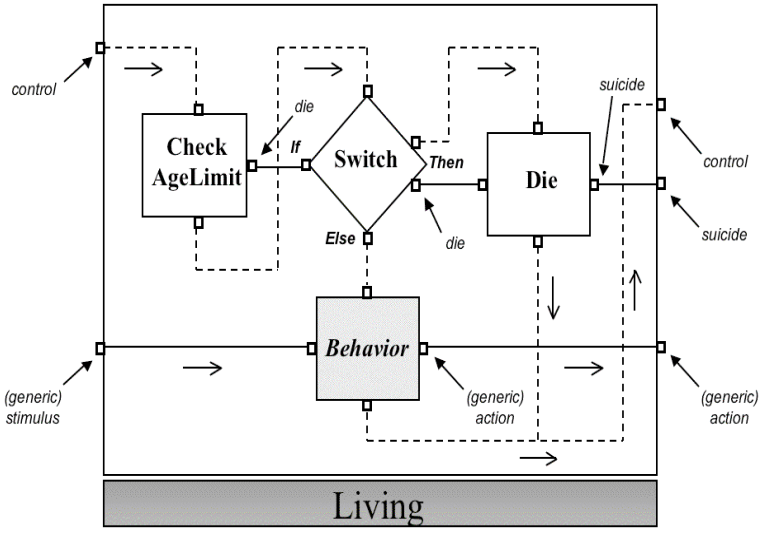


Fig. 11. `Living` abstract behavior

(in practice, it may e.g., remove the agent from the environment). Otherwise, `Behavior` is activated by `Switch`.

This design is a simplified form of a *design pattern* [11].[10] `Living` implements that pattern as some *"mini black-box framework"*, where the *hot spot* is the abstract component `Behavior`. To construct a specific agent behavior, we replace (instanciate) the abstract component `Behavior` with a concrete behavior, e.g., specific to an ant, egg, larva or queen [15].

5.2 The Behavior of an Ant

A worker ant has a relatively complex behavior because its various types of activities: move, pheromone following, egg carrying, egg caring. It is simplified in this paper. First, we instanciate `Living` into a concrete behavior specific to ants, named `Ant`. In practice, `Behavior` is replaced by a concrete behavior, named `AntActivity`.[11] Result is shown at Figure 12.

We now define the specific behavior of the ant, named `AntActivity`, shown at Figure 13. The ant explores its surroundings through a random movement (`Exploration` behavior), unless it perceives some stimulus (`ManageStimulus`

[10] We have identified others, e.g., *"exploration unless perception"*, used by the `Prey` behavior, in Section 4, and which will be reused for the `AntActivity` ant internal behavior, in Section 5.2. A further discussion about MALEVA design patterns may be found in [15].

[11] One may note that `AntActivity` has an additional output data port, in order to distinguish the two possible outputs: action (e.g., leave a pheromone or take food) and move, and their associated effectors and types. An alternative simplification is to consider a single output data port including all types of actions (including move).

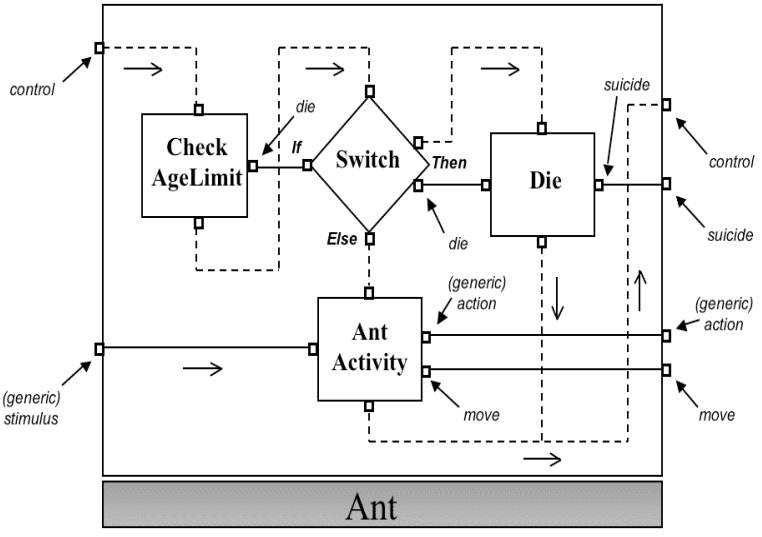


Fig. 12. `Ant` behavior

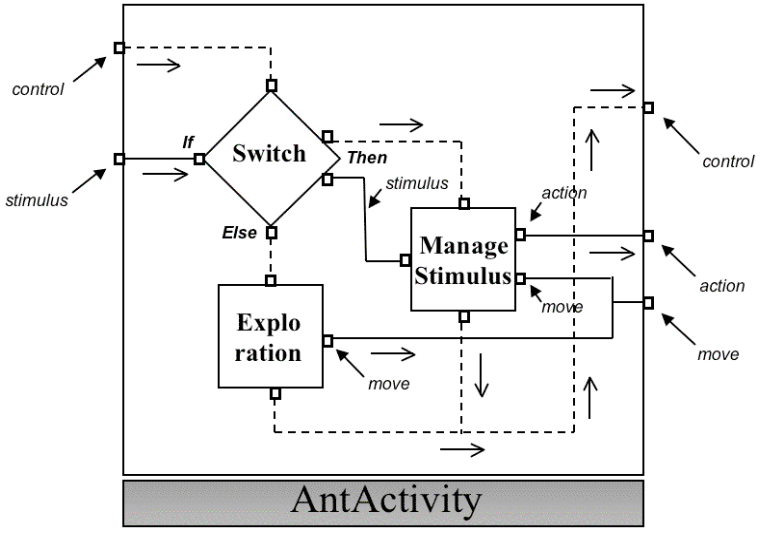


Fig. 13. `AntActivity` behavior: first level of decomposition of `Ant` behavior

behavior). Thus, `AntActivity` reuses the *"exploration unless perception"* pattern, already used for `Prey` and `Predator` behaviors (see Figures 9 and 10).

Because of space limitation, we do not detail the design of `ManageStimulus` behavior (follow gradient and take action when reaching a local maximum).

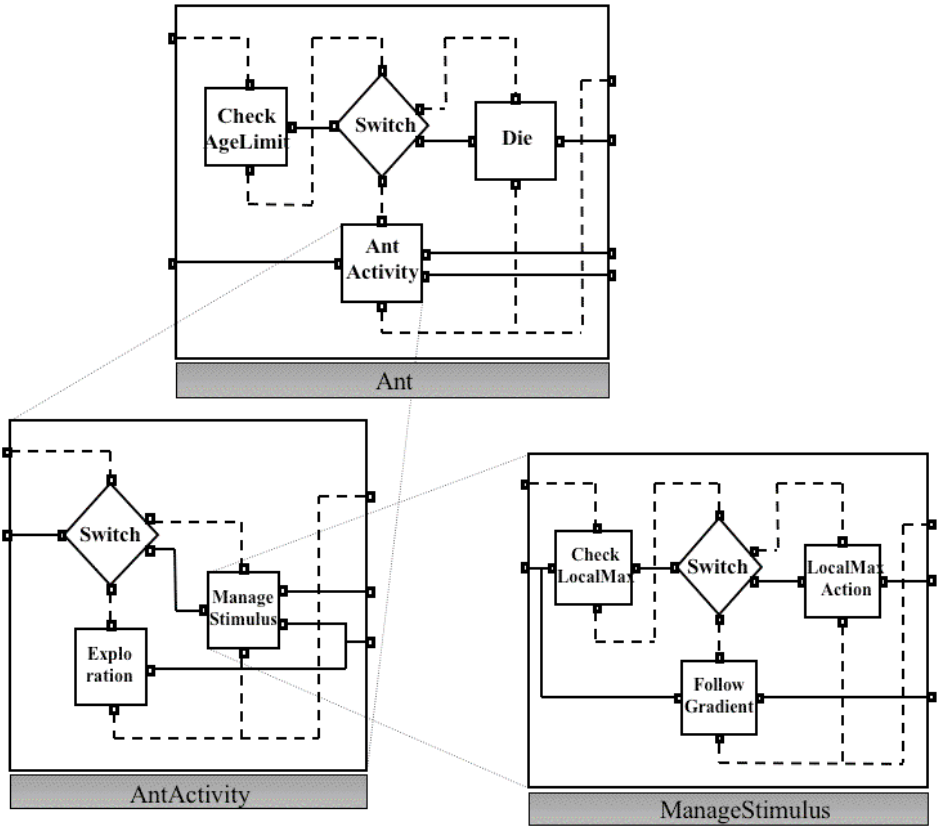


Fig. 14. `Ant` behavior: complete decomposition

The complete architecture of the ant behavior (including 3 levels and 14 components) is shown at Figure 14.

6 Implementation

6.1 Evolution of Implementation

The MALEVA component model and its associated prototype CASE tool have been implemented successively in three versions and languages: Delphi, Java [15], and C++ [16].

The Java-based reimplementation of MALEVA added typing to the components ports and connexions. This turned out to be useful for verifying interface compatibility between components. In addition, sub-typing helps at defining more abstract components. Java also supports inspecting various information about a component, thanks to its introspection facilities (API and tools). Thus, the designer can easily query a component to obtain its internal information.

The Java implementation, actually based on JavaBeans, also gave opportunity to compare our MALEVA prototype component model with an industrial component model. Note that the JavaBeans model conforms to a publish/subscribe communication model, *but* the implementation still relies on standard method call. In our implementation of MALEVA, a mailbox (FIFO queue of messages) is associated to each input data port and to the input control port, in order to decouple data transfer and actual activation.

6.2 Modes of Activation and Scheduling

At the level of the general scheduler, two alternative modes (or approaches) of activation have been implemented: an *asynchronous* mode and a *synchronous* mode. In the *asynchronous mode*, the different agents (and components) evolve independently. It may be more efficient, specially in the case of distributed implementation. Meanwhile, unless the designer also uses explicit control connexions between agents, the different agents may not be synchronised (some can compute ahead of others), depending on their relative processing speed. In the *synchronous mode*, the scheduler sends next activation trigger once all behaviours have finished, which ensures but also imposes a global synchronization. The choice between the two modes depends on the requirements for the application (see, e.g., [14] and [16] for more discussion).

6.3 From Methods to Components

The MALEVA prototype CASE tool includes a library of components (behavioral components and control components) ; an editor of connexion graphs (named CGraphGen, which stands for *concurrent graph generation*) ; a graphical environment for constructing virtual environments for situated agents ; and a run time support for scheduling and activating agents.

An interesting feature of CGraphGen [16] is the importation of actual Java code and its reification into MALEVA components. The granularity considered is a Java method. After specifying the class and method name, and its signature, CGraphGen automatically generates a corresponding component whose data ports correspond to the method signature: one input data port for each parameter, and one optional output data port for the result (none in the case of `void`). Two control ports (one input and one output) are also implicitly added. CGraphGen allows graphical connexion of both data-flow and control-flow between components, and the creation of composite components.

7 Related Work

In addition to the agent architectures already discussed in Section 2, we now quickly refer to a few additional related works, still focusing on the agent architectures offering some modular or compositional support at the level of one agent. See also, e.g., [2], for a recent more general survey of languages, architectures and platforms for multi-agent systems.

Like MALEVA, JAF (Java Agent Framework [13]), also based on JavaBeans, uses components to decompose behaviours of agents. JAF does not explicitly separate control flow from data flow. But it proposes some interesting matchmaking mechanism, where each component specifies the services that it requires. At component instantiation time, JAF looks for the best correspondence between the requirements specification and the components available. Another difference between JAF and MALEVA is at the level of behaviour decomposition. JAF decomposition appears at a relatively high level, whereas MALEVA promotes a fine grain behaviour decomposition, and its management through explicit control.

The MaSE methodology [9] includes a modular representation of agent behaviours as sets of concurrent tasks. Each task is described as a finite state automaton and is implemented as an object with a separate thread. A task can communicate with other tasks, inside the same agent, or with a task of another agent, through event communication. A first difference with MALEVA is that the implementation of MaSE concurrent tasks does not use components with explicit input and output ports. Another difference is that MALEVA provides more explicit control of activation, whereas MaSE concurrent tasks rely partly on some implicit control (inter-tasks implicit concurrency and synchronous message reception discipline). That said, as the MaSE methodology is actually very general, we could imagine using several of the MaSE steps to produce MALEVA components.

The DESIRE methodology and component model [3] is more high-level and knowledge-oriented than MALEVA and is more aimed at cognitive agents. It is based on a formal description considering separately a process/component level and a knowledge level. This approach enables some possibilities of verification, but at the cost of some added complexity in specifications. As opposed to MALEVA, DESIRE does not provide a fine grained control model for components.

8 Further Issues and Future Directions

A first issue, for our current component architecture, is that simulation designers must design activation models (control flow) from a relatively low-level perspective, with explicit manipulation of connexions. Abstract components and their related design patterns help at capitalizing and reusing experiences. While several patterns and reusable abstracts components have been tested and documented in various experiments (see, e.g., in Section 5 and also in [15]), we are still far from a complete library of such reusable activation patterns. We wish to provide the designers with several types of libraries: behavior components (e.g., `Exploration`) ; abstract components, e.g., `Living` ; control components, e.g., `Switch` ; and "system" components, e.g., for perception (sensors), action (effectors), inter-agent communication, migration. In addition, the component-based design of agents and the support of CASE tools using possible information (e.g., typing) should help in assisting the designer to analyse existing designs and to create new ones.

A second issue is that the experience with the specification of control through connexions shows that, in case of large applications, the connexion graphs may become large, *although* they may be hierarchical and encapsulated in composite components (e.g., see the recursive design of an ant behavior in Section 5). Some radical alternative approach to reduce the control graph complexity, and also to make it more accessible to formal analysis, is to abstract it in an adequate formalism. We think that a process algebra (such as CCS) [17] could allow the concise representation of complex activation patterns. The idea is somehow analog to coordination languages, but for very fine grained components. The starting point is to model data used for control (e.g., presence of prey, of predator, of pheromone...) as channels and synchronize activity of behaviors on them. The result is a compact term to express a control graph analog to the example of prey and predator (in Section 4):

$$isPrey.Follow \parallel isPredator.Flee \parallel (isNoPrey.Exploration + isNoPredator.Exploration)$$

where *isPrey*, *isPredator*, *isNoPrey* and *isNoPredator* are channels, connected to the sensors of the agent; and *Follow*, *Flee*, and *Exploration* are processes representing behaviors. Such formal characterization would also allow the semantic analysis of such specifications, for example through model checking.

A third issue is the dynamicity of behaviors. An example of modeling is the metamorphosis process of ants (egg, larva, ant), introduced in Section 5. Current implementation strategy relies on a specific meta-component to manage the reconfiguration and reassemblage of behaviors. We are currently considering using a higher level mechanism, based on concepts of configurations, roles and policies, such as [12]. Last, to allow the dynamicity of formalisms for activation patterns, we are considering models of process algebras supporting dynamicity and channel name passing, such as the Pi-calculus [18].

9 Conclusion

In this paper, we presented some experience in using a component model to design and implement agents. This model is relatively original in the explicit management of activation control through control ports and connexions, by applying the concept of component to the specification of control. Several experiments illustrate how MALEVA can support various forms of potential reuse through: structural composition of behaviors, abstract components and design patterns, and specialization of intra-agent scheduling policies (that latter issue is discussed in [6]).

Considering rationales for agent architectures, we believe that there is no ultimate *best* agent architecture, as it depends on the application domain and requirements. General purpose (also named hybrid) architectures, like InteRRaP, which attempt at reconciling both cognitive and reactive architectures, turn out to be powerful, but also complex. On the contrary, our architecture focuses on a lower-level agent component model with a fine-grained control. It was initially

more targeted at reactive agent models for multi-agent simulation, but we believe that the MALEVA component model is more general, the issue being more in providing sufficiently rich libraries of components and abstract architectures, supporting the types of architectures and applications targeted (e.g., interaction protocols for e-commerce, reasoning components for rational/cognitive agents, etc.). More generally speaking, we believe that some features of our architecture model may be transposed, and that making control available at the composition level may help the use of components within frameworks of applications vaster than those in which they had been initially thought.

Acknowledgement

We would like to thank Marc Lhuillier, Alexandre Guillemet and Grégory Haïk, for their contribution to the MALEVA project.

References

1. F. Bellifemine, A. Poggi, G. Rimassa, Developing Multi-Agent Systems with a FIPA-compliant Agent Framework, *Software Practice and Experience*, (31): 103–128, 2001.
2. R. Bordini, L. Braubach, M. Dastani, A. El Fallah Seghrouchni, J.J. Gomez-Sanz, J. Leite, G. O'Hare, A. Pokahr, A. Ricci, A Survey of Programming Languages and Platforms for Multi-Agent Systems, *Informatica*, 30:33–44, 2006.
3. F. Brazier, B. Dunin-Keplicz, N.Jennings, J. Treur, Formal Specification of Multi-Agent Systems : a Real-World Case, *1st International Conference on Multi-Agent Systems (ICMAS'95)*, San Francisco, CA, USA, MIT Press, 1995, pp. 25–32.
4. F. Brazier, C. Jonker, J. Treur, N. Wijngaards, Compositional Design of a Generic Design Agent, *Design Studies Journal*, (22):439–471, 2001.
5. J.-P. Briot, Composants logiciels et systèmes multi-agents, *Technologies SMA et leur utilisation dans l'industrie*, A. El Fallah-Seghrouchni (ed.), Collection IC2, Hermès/Lavoisier, France, to appear in 2007.
6. J.-P. Briot, T. Meurisse, A Component-based Model of Agent Behaviors for Multi-Agent-based Simulations, *7th International Workshop on Multi-Agent-Based Simulation (MABS'06)*, AAMAS'2006, Japan, May 2006, pp. 183–190.
7. R.A. Brooks, A Robust Layered Control System for a Mobile Robot, *IEEE Journal of Robotics and Automation*, 2(1):14–23, March 1986.
8. E. Bruneton, T. Coupaye, M. Leclerc, V. Quema, J.-B. Stefani, An Open Component Model and its Support in Java, *7th International Symposium on Component-Based Software Engineering*, No 3054, LNCS, Springer-Verlag, May 2004, pp. 7–22.
9. S.A. DeLoach, Multiagent Systems Engineering: A Methodology and Language for Designing Agent Systems, *Agent-Oriented Information Systems (AOIS'99)*, Seattle, WA, USA, May 1999.
10. A. Drogoul, B. Corbara, D. Fresneau, MANTA: Experimental Results on the Emergence of (Artificial) Ant Societies, in *Artificial Societies: the Computer Simulation of Social Life*, N. Gilbert and R. Conte (eds), UCL Press, U.K., 1995.
11. E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns - Elements of Reusable Object Oriented Software*, Addison-Wesley, 1995.

12. G. Grondin, N. Bouraqadi, L. Vercouter, MaDcAr: an Abstract Model for Dynamic and Automatic (Re-)Assembling of Component-Based Applications, *9th International SIGSOFT Symposium on Component-Based Software Engineering (CBSE'2006)*, No 4063, LNCS, Springer-Verlag, 2006, pp. 360–367.
13. B. Horling, A Reusable Component Architecture for Agent Construction, *Technical Report* No 1998-49, Computer Science Dept., UMASS, MA, USA, October 1998.
14. B. G. Lawson, S. Park, Asynchronous Time Evolution in an Artificial Society Mode, *Journal of Artificial Societies and Social Simulation*, 3(1), 2000.
15. T. Meurisse, J.-P. Briot, Une approche à base de composants pour la conception d'agents, *Journal Technique et Science Informatiques (TSI)*, 20(4):583–602, Hermès/Lavoisier, France, April 2001.
16. T. Meurisse, Simulation multi-agent : du modèle à l'opérationnalisation, *Thèse de doctorat (PhD thesis)*, Université Paris 6, Paris, France, July 2004.
17. R. Milner, *A Calculus for Communicating Systems*, Springer-Verlag, 1982.
18. R. Milner, *Communicating and Mobile Systems: the Pi-Calculus*, Cambridge University Press, 1999.
19. J.P. Müller, M. Pischel, The Agent Architecture InteRRaP: Concept and Application. *Technical Report* RR-93-26, DFKI, Saarbrucken, Germany, 1993.
20. J.P. Müller, Control Architectures for Autonomous and Interacting Agents: A Survey, In *Intelligent Agent Systems: Theoretical and Practical Issues*, No 1209, LNAI, Springer-Verlag, 1997, pp. 1–26.
21. P.-G. Ricordel, Y. Demazeau, Volcano, a Vowels-Oriented Multi-Agent Platform, *2nd International Workshop of Central and Eastern Europe on Multi-Agent Systems (CEEMAS'01)*, No 2296, LNCS, Springer-Verlag, 2001, pp. 253–262.
22. S. Moss, P. Davidsson (eds), Multi-Agent-Based Simulation, *2nd International Workshop on Multi-Agent Based Simulation (MABS'2000) - Revised and Additional Papers*, No 1979, LNCS, Springer-Verlag, 2001.
23. M. Shaw, D. Garlan, *Software Architectures: Perspective on an Emerging Discipline*, Prentice Hall, 1996.
24. Y. Shoham, Agent Oriented Programming, *Artificial Intelligence*, 60(1):51–92, 1993.

Part II

Comparing Apples with Oranges: Evaluating Twelve Paradigms of Agency

Linus J. Luotsinen, Joakim N. Ekblad, T. Ryan Fitz-Gibbon, Charles Andrew Houchin, Justin Logan Key, Majid Ali Khan, Jin Lyu, Johann Nguyen, Rex R. Oleson II, Gary Stein, Scott A. Vander Weide, Viet Trinh, and Ladislau Bölöni

School of Electrical Engineering and Computer Science
University of Central Florida
Orlando, FL

Abstract. We report on a study in which twelve different paradigms were used to implement agents acting in an environment which borrows elements from artificial life and multi-player strategy games. In choosing the paradigms we strived to maintain a balance between high level, logic based approaches to low level, physics oriented models; between imperative programming, declarative approaches and "learning from basics" as well as between anthropomorphic or biologically inspired models on one hand and pragmatic, performance oriented approaches on the other.

Instead of strictly numerical comparisons (which can be applied to certain pairs of paradigms, but might be meaningless for others), we had chosen to view each paradigm as a methodology, and compare the design, development and debugging process of implementing the agents in the given paradigm.

We found that software engineering techniques could be easily applied to some approaches, while they appeared basically meaningless for other ones. The performance of some agents were easy to predict from the start of the development, for other ones, impossible. The effort required to achieve certain functionality varied widely between the different paradigms. Although far from providing a definitive verdict on the benefits of the different paradigms, our study provided a good insight into what type of conceptual, technical or organizational problems would a development team face depending on their choice of agent paradigm.

1 Introduction

Researchers have designed a bewildering variety of paradigms for the control of agents. Even if we restrict our inquiry to the case of embodied agents, that is, artifacts which operate either in the physical world or a simulation of it, virtually every paradigm of artificial intelligence, software engineering or control theory was deployed with more or less success. However, wide ranging comparisons of agent paradigms are rare. When new methods and paradigms are introduced, they are compared with only several, closely related approaches which

R.H. Bordini et al. (Eds.): ProMAS 2006, LNAI 4411, pp. 93–112, 2007.

are considered direct competitors of the proposed paradigm. Making or revisiting comparisons between paradigms is a controversial, difficult and hard-to-sell work. One might argue that a researcher might better spend his or her time in designing new paradigms or improving existing ones instead of comparing, say, swarm algorithms with affective computing in the design of embodied agents. There might be people offended by the results, with reasonable claims that the methodology was incorrect, the implementation of the paradigm substandard, or simply, the measured quantity is not relevant to the given paradigm.

The fundamental question, of course, is whether if any of these comparisons make sense. We argue that **if both paradigms A and B can be used in the implementation of the same requirements, then these two paradigms can (and indeed, should be) compared**. That is not to say that the comparison is easy or that it can be reduced to a single numerical "score". Different paradigms have different strengths and weaknesses, and the goal of a comparison study is to shed light on these differences. Although we do not expect definite answers on questions like "which paradigm would eventually lead to an agent passing the Turing test", we can provide insight into lesser but still important questions such as:

- Would the implementation provide adequate performance?
- Can a rigorous software engineering process be applied to the development?
- Can the performance be predicted?
- Can human expertise in the problem domain be transferred to the agent?
- What will the development effort be?
- Will the resulting agent be predictable in its actions?

The remainder of this paper is organized as follows. In Section 2 we present the Feed-Fight-Multiply game, our control problem. We succinctly describe the twelve agents we implemented in Section 3. In Section 4 implementation effort and complexity measurements are presented for each agent. We detail our findings in Section 5.

2 The Feed-Fight-Multiply World

To study the benefits and drawbacks of various agent paradigms, we decided to place them in a virtual environment in which many of the real world challenges are reflected. We did not choose one of the existing environments, because the existing implementations would have skewed the result of the comparison. One requirement towards the environment was the existence of *multiple paths to success*. We expected that agents implemented in various paradigms will have a different external behavior as well. By measuring success as the conformance to a predefined behavior we would have favored some paradigms and disadvantaged others. In addition, having multiple paths to success is a quality of most natural environments and many artificial ones [6].

Upon these considerations, we implemented the Feed-Fight-Multiply (or Mate) game, which borrows elements from turn-based multi-player strategy games and

artificial life. Agents are sharing a two-dimensional environment having accessible zones and obstacles. The agents can sense their environment within the range of their *sensors*. Food resources appear at random points in the environment; consuming food increases the energy level of the agents. Finally, agents can multiply by (non-sexual) reproduction. The environment was implemented in the YAES simulation environment [3]. Figure 1 shows a typical FFM game in progress.

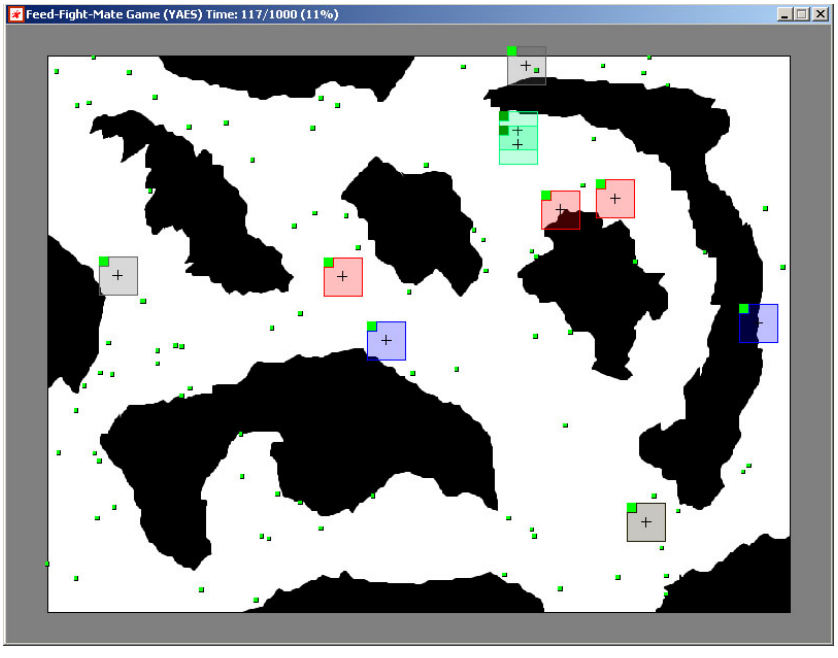


Fig. 1. Screenshot of the Feed-Fight-Multiply environment

An additional concern was to choose the level of the services provided by the environment. Evidently, natural environments do not provide any kind of service, but this would make the implementation of the agents unduly difficult. The guiding principle was that whenever the problem had a well known, standard implementation, we had chosen to implement it in the environment, and provide it as a service to the agents. These services included: the scanning of the sensor range for agents and food, tracking of moving agents and identifying agent types.

Finally, instead of keeping a single score, we decided to record multiple parameters of the agent behavior. This meant that not only there were multiple paths to success, but the final goals of the agents could be different as well. Of course, all agents were required to work towards their survival, but besides that, the criteria for success could be maximum amount of resources gathered, survival on minimum amount of resources, largest number of agents killed, number of individual agents of the same type at the end of the game, or others.

3 Twelve Agents, Twelve Paradigms

We have developed twelve agents, implemented in twelve different paradigms of agency. In choosing the paradigms we strived to maintain a balance between high level, logic based approaches and low level, physics oriented models; between imperative programming, declarative approaches and "learning from basics" as well as between anthropomorphic or biologically inspired models on one hand and pragmatic, performance oriented approaches on the other. The implemented agents are concisely described in Table 1. The developers were instructed to develop paradigm-pure implementations and to design the agents such that the "spirit" of the paradigm is best expressed. When the paradigm could cover only some of the required functionality, the developers could use some limited heuristics.

Table 1. Concise description of the twelve implemented agents

Name	Paradigm	Paradigm coverage	Team-work	Offline Learning	Realtime adapt.
`AffectiveAgent`	Affective model, anthropomorphic lifecycle	Limited	No	No	Yes
`GenProgAgent`	Genetic programming	Full	Yes	Yes	No
`Reinforcer`	Reinforcement learning	Full	Yes	Yes	No
`CBRAgent`	Case based reasoning	Full	No	Yes	Yes
`RuleBasedAgent`	Forward reasoning	Full	Yes	No	No
`NaiveAgent`	Naïve programming (scripting)	Full	Yes	No	No
`GamerAgent`	Game theory	Limited	Yes	No	No
`CrowdAgent`	Crowd model	Limited	Yes	No	No
`NeuralLearner`	Neural networks	Full	No	Yes	No
`SPFAgent`	Social potential fields	Limited	Yes	No	No
`CxBRAgent`	Context based reasoning	Full	No	No	No
`KillerAgent`	Simple heuristics, with path-planning	Full	No	No	No

3.1 AffectiveAgent: Anthropomorphic and Affective Model

The basic premise behind the affective agent paradigm is that the agent behaves with an emotional frame of reference with which to weight its decisions. Besides providing the agent with emotional states such as anger, contentment or fear, we also made it to mimic the basic lifecycle of humans: agents have a childhood, maturity and old age, with their corresponding goals and priorities. In broad

lines, our implementation is an adaptation of the agents from [17]. The affective model plays two roles in the behavior of the agent: *action selection* (e.g., what to do next based on the current emotional state) and *adaptation* (e.g., short or long-term changes in behavior due to the emotional states).

The short term variables which control the behavior of the agent are the *action tendency* and the *conflict tendency*. The dynamic action tendency is the probability whether an agent will fight or flee in a given situation. To adapt the action tendency to the outcome of the agent's interactions, the action tendency is updated by the *adaptation rule* depending whether the agent is experiencing loss or success.

This dynamic action tendency is used in calculating another dynamic parameter of the adaptive agents, namely, conflict tendency. This parameter determines whether an agent seeks conflicts or avoids them. Emotive states such as anger or fear are determined in terms of ranges of the action and conflict tendency.

Besides the mood of agent, its behavior is determined by its age. The agent remains content, without adaptation, until the agent comes of age, which is set, with apologies to Tolkien, to 33 cycles. After the age of maturity, the agent's conflict tendency is adapted every 10 cycles. To avoid agents becoming bogged down in an emotional quagmire, a catalyst was installed in the way of a mood swing. At 50 cycles and every 25 subsequent cycles, the agent's current action tendency is randomly reset to a new value and then action tendency and conflict tendency are recalculated. This provides a potentially dramatic change in mood. An agent could easily shift from an action tendency of 0.8 angry to 0.3 fearful.

The age of maturity was also employed to delay the agent's mating. Moreover, the agent's mating is also limited by mood and by energy level. An agent that is angry cannot mate. Only an emotional state of fearful or content will allow the agent to mate.

3.2 GenProgAgent: Genetic Programming

Genetic Programming (GP) [10] is an evolutionary algorithm in which the evolutionary units are computer programs or functions described by tree structures consisting of conditional branches, mathematical operators, variables and constants [2]. We based `GenProgAgent` on the generational genetic algorithm [8].

The evolution of the behavior of the agent is split in two stages. In the first stage we evolve *tactical behaviors*, which control primitive actions such as eat food, explore, attack and multiply. In the second stage we evolve *game strategies* by combining the behaviors from the first stage using Finite State Machine (FSM) structures.

Stage 1 - Evolving Tactical Behaviors. Four types of primitive behaviors were created: Eat-food, Explore, Attack and Flee. The eat-food behavior was generated using a fitness function which defines an ideal individual as an agent that does not collide with obstacles and that eats all the available food resources. The fitness is derived based on the number of failed move sequences, the length of the failed move sequences and the amount of food eaten. Although the algorithm

did not find an optimal solution, the best individual was able to, in most cases, effectively avoid obstacles and consume available food resources. The behaviors for Explore, Attack and Flee can be evolved similarly, or can be created from the Eat-Food behavior by replacing the heuristics. As we did not manage to evolve optimal primitive behaviors which reliably avoided being stuck on obstacles, we decided to augment the evolved behaviors with helper heuristics. The heuristics used provide directions to closest food, opponent agent and unexplored areas using A* search.

Stage 2 - Evolving Game Strategies. With the tactical behaviors already created, the next challenge is to decide which tactics to be applied at any given moment. Tactical decision are considered the states of a finite state machine, and we apply genetic programming to evolve the transition rules for these structures. Two types of game strategies were created: Balanced and Aggressive. The balanced strategy seeks to create an agent that doesn't specialize on any type of behavior. The aggressive strategy seeks to create an agent that specialize on attacking and killing other agents. The game strategies were generated in a FFM game with 6 additional opponent agents. The purpose of the opponent agents is to generate hostile and conflicting situations from which strategies resolving these situations can be evolved.

To validate the game strategies that were evolved, we allowed them to execute in the FFM game for 20 individual runs each of 4000 simulation cycles. Each run was set up with 5 balanced strategy agents, 5 aggressive strategy agents, and 6 opponent agents.

Results show that the aggressive strategy was able to perform on average 23.45 successful attacks. This is more then twice as much as the balanced strategy was able to do. The aggressive strategy involves no eating of food and few attempts to flee when in conflict. The balanced strategy has on average 154.04 successful eat attempts, map coverage of 23% and 10.57 successful attack attempts. The balanced strategy is, as expected, more general then the attacker strategy as it involves eating, attacking and more extensive exploring.

3.3 Reinforcer: Reinforcement Learning

Reinforcement Learning (RL) has the ability to learn in an unknown domain without prior knowledge [13]. The specific technique chosen for the `ReinforcementAgent` was Temporal Difference (TD) learning. This approach uses a table of state action pairs and their corresponding reward. If an action leads to a good result, but this is not detected until several steps later, that good result will immediately propagate back to the initial action and therefore favor it in the future. The following formula was used in the `ReinforcementAgent` implementation.

$$Q(s_t, a_t) = (1 - \alpha) * Q(s_t, a_t) + \alpha * (r_t + \gamma * max(Q(s_{s+t}, a_{t+1}))) \quad (1)$$

$$\alpha = 1/(1 + visits(s, a)) \quad (2)$$

Where α is related to the number of times that the state action pair has been visited, γ is a user defined value between 0 and 1, $Q(s, a)$ and $r(s, a)$ is the value and reward for current state action pair respectively.

The action set for `the ReinforcementAgent` is defined by 20 movement actions. Actions for eat, attack and flee are also included in the action set. The state set consist of various energy level thresholds, possible actions to take in each game round and on the objects and their directions as seen in the agent's sensors. In total there are 117760 state action pairs. Reinforcement is applied using direct stimuli from the environment. Negative reinforcement is imposed when the agent fail to perform some action or when the agent performs to many consecutive actions of equal type. In similar manner, positive reinforcements is given when an agent successfully performs some action.

The temporal depth level, or action chain depth, was empirically set to 5. This value was chosen for several reasons. First, it needed to be greater than four since the method of successive actions that returned to the previous state was penalized in this system. Second, four moves and eat was the number of actions required at a maximum to eat most food in the local map sensor range. Finally, it was experimentally determined to be a good balance of delaying reward that seemed to make sane decisions that were not too local and much longer the global reward seemed to not converge easily.

3.4 CBRAgent: Case-Based Reasoning

Case-Based Reasoning (CBR) [1,16] is the process of intelligently solving new problems based on the previous similar problems. The basic steps of CBR are *retrieve*, *reuse*, *revise* and *retain*. First, CBR retrieves the most relevant case to the current problem at hand. The retrieved case is reused and revised to incorporate minor variations in the solutions. This adaptation step gives CBR a power to form more precise and accurate solutions to the future problems. Finally the revised solution is retained for future use. [11] has shown the capability of CBR as the intelligent search method for controlling the navigation behavior of the autonomous robot. Our implementation is an adaptation of this model.

The case was represented using ten parameters (the closest food, enemy and obstacle, the density of food and enemies, the ratio between the agents health and the opponents, and parameters determining the actions currently available to the agent). A weighting scheme was used to emphasize the more important features. We have identified 19 historic cases that are selected based on the performance of the agent in training simulations. Even with the small number of cases, the CBR-based agent shows drastic improvement over the random behavior and can be refined by adding more specific cases to the library. The selection of the case was done with a distance matrix based selection. The reliability of the case based on the previous failures is used as a weight in order to encourage the selection of variety of cases. The reliability of the case is dynamically adapted depending on whether the case can or cannot be successfully applied to the current situation.

After the action to be performed is selected by the CBR module, a set of heuristics are used to adapt the action to the current environment. First, the

heuristic module checks whether the proposed action is feasible. In case of failure, the reliability measure of the case which proposed the action is reduced. The reliability of the case is used as a weight during the calculation of the distance matrix, so that case with the less reliability will have greater distance in the subsequent cycles. The second heuristic calculates the direction and the speed of the agent movement. For instance, eating a food item requires the direction toward the food and appropriate setting of the speed such that the agent stops at the food item.

3.5 RuleBasedAgent: Forward Reasoning

A rule based system consists of an inference engine and knowledge base. The engine's reasoning mechanism uses a forward-reasoning technique. The knowledge base contains a fact base and a rule base. The fact base acts as a repository of all the truths that is seen or understood by the agent. The fact base is periodically updated with new sensor data, which triggers the execution of rules. To reduce the number of rules necessary to determine the behavior of the system, we have decided to choose the atomic actions at a relatively high level of abstraction. For example, a typical rule would have the consequent of movement towards a particular spot. Many additional intermediate rules could have been implemented to determine exactly how to move towards the objective. Instead, once this rule has been fired, a helper function is called to determine where the objective is and binds various directional parameters of the consequent of the rule.

The utilization of these helper functions reduces the number of intermediate rules that are necessary in the rule base and allows the developer to concentrate on the upper level behavioral aspects of the agent in the rule base.

The ease of modification of the rules also allows the developer to quickly adjust the behavior of the agent. This is primarily done through the change in salient values of each rule. In addition to salient values, the speed, aggressive nature, and other parameters are adjusted to yield an agent with completely different behavior traits without any modification to the main engine of the system or major change to the structure of the rules. The value of this property is that major alterations can be made to the agent's behavior with little modification to the actual structure and knowledge. A result of this is an increase of time dedicated to testing and tuning the agent for performance.

The rule base of the `RuleBasedAgent` agent consists of 15 rules. The consequent of each rule may result in another fact pushed onto the fact base or an action of the agent which would terminate the inference mechanism for that simulation cycle. The *salience* of the rule is used to aid in conflict resolution as well as the method of sorting in the rule base stack. This method prioritizes the various rules which in turn define the behavior of the agent.

The `RuleBasedAgent` implements all the five basic commands of the game (movement, feeding, fleeing, attacking and mating). Whenever a decision needs to be taken, it is determined by synthetic facts in the fact base of the agent. For instance, the choice to attack is determined by the fact AGGRESSIVE, while a choice to flee is triggered by the fact TIMID. Whenever there is a potential for

an encounter with another agent, the decision of the aggressive or timid behavior is made by the relative energy of the agents.

The rule based agent implements flocking behavior with other rule based agents in its sensor range. If the fact base contains the synthetic fact FLOCK, the move commands will be restricted to movements which allow the agent to remain in the flock. The leader of the flock is the agent with the lowest id (the oldest agent). Two scenarios occur when food is within range while in a flock. If the GREEDY fact exists, the agent attempts to move and acquire the food closest to them. If the fact does not exist, then the agent only attempts to move and acquire a food resource if no other rule-based agent is closer to the resource. The second scenario promotes efficient feeding while in the pack to avoid agents' ineffective attempts to acquire the same resource. The agents in the pack also synchronize their attack and fleeing behaviors. If the leader of the pack attacks an agent, all the members of the pack will attack, regardless of their energy levels or aggressiveness ratio.

3.6 NaiveAgent: Naive Programming, Scripting

Naive programming is a style of coding that allows the developer to hand-optimize the code for a particular task. `NaiveAgent` relies on the hand scripting of encounters for its success - a technique frequently used in the development of multi-player games. For each possible encounter, a script was written specifying how the agent should react. In the following we discuss some of the heuristics used in the implementation:

Exploration: in the absence of other tasks, the `NaiveAgent` moves around the environment with its maximum speed. This way, by covering more area, the probability of finding food increases. It was found that the benefit of finding more food outweights the extra expenditure of energy. The higher coverage also increases the chance of encountering other agents, which is beneficial, given the aggressive nature of the `NaiveAgent`.

Obstacle avoidance: Instead of using a sophisticated decision-making process to guide the movement of the agent, the `NaiveAgent` simply moves right every time it encounters an obstacle. To avoid getting stuck, a `failcount` variable is incremented each time the agent makes a right turn. Only encountering another agent or food particle can reset the fail count. If the fail count is greater than five, the direction is chosen randomly.

Social behavior: If the `NaiveAgent` has a particle of food and another `NaiveAgent` is within the sensor range, only the agent with the lowest energy level is allowed to eat.

Aggression: Whenever a different agent is detected in the sensor range, and the agents' energy level is larger than 120% of the opponents, the `NaiveAgent` attacks the opponent. If more than two agents are in the range, the `NaiveAgent` attacks the weakest opponent.

3.7 GamerAgent: Game Theory

Game theory [9] is a mathematical formulation of cooperative or competitive interaction between multiple entities. The key concern in game theory is to extract rational (optimal) behavior from a given interaction between autonomous agents. We model the FFM world as a zero-sum game.

The game consists of two entities, and each one of them can choose from two strategies: attack or flee. The utility functions for each strategy are based on the ratio of energy levels Δ and the likelihood of attack or flee by the other agent based on previous interactions μ. We will denote $U_{a,b}$ the utility of taking action a when the opponent agent takes action b (where the actions can be A for attacking and F for fleeing. The utility functions are defined as follows:

$$U_{A,A} = (1 - \mu) \times 100 + \Delta \times 200 \tag{3}$$

$$U_{A,F} = \mu \times 100 + \Delta \times 200 \tag{4}$$

$$U_{F,A} = \mu \times 100 - \Delta \times 200 \tag{5}$$

$$U_{F,F} = (1 - \mu) \times 100 - \Delta \times 200 \tag{6}$$

Given the matrix, the optimal strategy is chosen by summing up the utility for each strategy. The strategy that provides the maximum utility is then chosen as the optimal strategy.

Several heuristics were used to guide the agent to explore the map and eat available food. The expert agent uses an internal data structure representing the perceived game map as a base for the heuristics.

Heuristic 1: IF the agent is not able to eat since last 20 to 100 simulations and there is no food or other agent in sight **THEN** move to the least explored direction on the map for 50 simulation steps.

Heuristic 2: IF If the agent is not able to eat since last 100 to 200 simulations and there is no food or agent in sight **THEN** obtain two random directions (which are non-opposite to each other), and move in those directions for 200 simulation steps.

Heuristic 3: IF there is more than one agent in the sensor range **THEN** flee in the direction which has the least number of agents.

Heuristic 4: IF the agent energy level reaches $35000 * mateEnergyFactor$ and there is no agent or food in sensor range **THEN** multiply and increase $mateEnergyFactor$ by 0.2.

Heuristic 5: IF food is visible in the sensor range and no agent is visible **THEN** approach the food and eat it.

3.8 CrowdAgent: Crowd Model

Crowd modeling techniques traditionally take inspiration either from fluid systems or particle systems. Both approaches deal with attractive, repulsive and frictional forces; in addition, particle systems place motion decision with the individual [4]. In the implementation of the `CrowdAgent`, we chose the aggression

level of the agents as the grouping characteristic of the crowd. An agent will start out with an initial aggression rating, $A(0) = A_i$, and then migrate towards the aggression level of the agents surrounding them. This transition is governed by:

$$A(t + \Delta t) = A(t) + \frac{(A(0) - A(t))^3 * \Delta t}{am} + \sum_{b=0}^{n} \left(\frac{(A_b(t) - A(t))}{cosh(A_b(t) - A(t))^2} \right) \cdot \frac{t}{max((D_b^2), 4)} \tag{7}$$

Where D_b is the distance between the current agent and agent b. The first term of the equation guarantees that if the agent is not surrounded by other agents it will return to its initial aggression level. If there are other crowd agents in the neighborhood, the agent will have its aggression level pulled towards the aggression level of each of the surrounding agents. The motion of an agent is related to the position of all other agents in the sensor range, and what there aggression levels are. The equation of motion used is:

$$X(t + \Delta t) = X(t) + \Delta t \cdot V_x \cdot \textstyle\sum_{b=0}^{n} \frac{pF_b \cdot (X(t) - X_b(t))}{max(D_b)^3), 1)} \tag{8}$$

A similar function is calculated for the Y direction. Once again we are summing over all agents in the sensor range, but this time we also generate a factor for the attraction between agents. The pF attribute is based on the aggression of the agent of interest and the aggression of the agent in the sensor range. This is an attractive/repulsive attribute which is defined by the piecewise function

$$f(n) = \begin{cases} -1 * \frac{pfA}{pfB^2} * |A - A_b|^2 + pfA \\ \quad \text{if } |A - A_b| <= pfB \\ pfC * 4 * \left(|A - A_b| - \frac{pfB + (10 - pfB)}{2} \right)^2 - pfC \\ \quad \text{if } |A - A_b| > pfB \end{cases} \tag{9}$$

The pF factor will give an attractive influence between 0 and pfB, the remaining distance will give a repulsive influence. As long as the attractive forces are not made too large then the individuals will have the ability to separate from a group, and rejoin another group.

As the particle grouping paradigm deals only with the motion of agent in the presence of crowds, it was supplemented by a series of heuristics. In the absence of other agents, the agent will perform random wandering. If food is detected, the agent moves directly towards the food. The agent is reproducing with a random probability whenever the energy level is high enough. A simple heuristic was used for fighting: the agent tries to avoid coming in contact with other agents, but if it comes into contact, it will attack. Finally, a simple heuristics is used for obstacle avoidance. If an obstacle is in the direction you are trying to move then keep turning to the right until you find an open direction and go that way.

In practice we found that there was a need for at least 4 agents of this type to get any really dynamic interactions going, and this was also the needed level to guarantee a long survival time, the algorithm performing the best with 6 agents of this type, given the limitations of the environment size.

3.9 NeuralLearner: Neural Networks

Neural networks are a natural choice as the control paradigm for embodied agents. An agent is trained with a set of training data representing sensory inputs and desired actions as outputs, and a learning algorithm such as back-propagation [15] is used to teach the agent the optimal behavior.

Other models, such as PolyWorld [18] used very general neural network structures and Hebbian learning. In many instances, the neural networks are used in combination with genetic algorithms or evolutionary programming [7]. Our implementation was based on a pure neural network approach, without heuristics or evolutionary programming.

The defining difficulty in our implementation was the acquisition of training data, a problem noticed by other artificial life researchers as well [19]. The problem is that there is no input-output mapping inherent to artificial life simulations. One must find a mapping that the neural network should estimate, and then acquire data based on that mapping. This requires the pre-existence of other agents, and the performance of the `NeuralLearner` will be determined by the performance of the model agent. Based on this balance, this project has two parts: search the entire input-output mapping space for a possible solution, then teach that solution to a neural network agent.

All the decisions in a `NeuralLearner` agent are made by a single multilayer neural network. The inputs to this network consisted of the agent's current energy level, the presence and direction of another agents, food and obstacles. Also included in the input was whether or not the agent could currently eat, mate, attack, or flee. The output of this network was an action selection (move, eat, attack, flee, mate), a direction (north, south, east, or west), and speed value. To acquire data for the training of the `NeuralLearner` a random agent was first created to explore the artificial life world and record data to be used to train the network. The actions of the random agent where filtered, and the training set contained only the input-output pairs that either led to a direct increase in energy, or kept the agent alive over a long period of time. Unfortunately, the random agent usually (about 80% of the time) made a decision that did not lead to useful data. Hence, the random agent approach was a very inefficient method of acquiring data. To improve data acquisition, the random agent was pushed towards situations where it would have experiences, both good and bad.

The resulting data sets were used to perform offline training on the neural network of the agent. The network was then used statically with the agent, that is, no more learning took place.

3.10 SPFAgent: Social Potential Fields

Social potential fields [14] are a way to control autonomous agents using inverse-power laws on attractive and repulsive forces between the agents and objects of the environment. We have implemented an agent whose movement is determined by a set of forces which attract or repulse the agent to agents and object of its sensor field. The resulting force is

$$\overline{F_i} = \sum_{j=1}^{N} \overline{v_{ij}} \cdot \left(-\frac{c_1}{r^{\sigma_1}} + \frac{c_2}{r^{\sigma_2}} \right) \tag{10}$$

where $\overline{v_{ij}}$ is the unit vector of the direction from agent i to agent j, and r is the distance from agent i to agent j. The parameters $c_1, c_2 \geq 0$ and $\sigma_1, \sigma_2 > 0$ are determining the nature of the forces between the agent and the object.

Once we decided on the general form of the forces, the next step is the choice of the parameters c_1, c_2, σ_1 and σ_2 such that the desired behavior of the agent is obtained. In practice, the determination of these parameters is a result of experience and experimentation. We have determined four sets of these parameters, which describe the relationship of a social potential field agent to (1) another social potential field agent, (2) an other agent, (3) food items and (4) obstacles. The experimentally obtained values are displayed in Table 2.

Table 2. The inverse power force law constants used in `SPFAgent`

Object of Interest	c_1	c_2	σ_1	σ_2
SPF Agent (a)	45.0	20.0	1.0	0.7
Other Agent (n)	45.0	0.0	1.0	0.7
Food (f)	0.0	20.0	1.0	0.8
Obstacle (o)	5.0	0.0	5.0	1.0

During testing, two major problems were found with the movements of the agents. Agents had a tendency to be stuck to into local minima, such as becoming immobile in the geometrical center of several food sources. Second, agents frequently overshot the food location and performed an oscillatory movement around it. A similar problem led to the agent bouncing indefinitely between two obstacles. These problems were solved by adding heuristics which (a) break the tie between the attraction forces and (b) prevent repetitive movements.

As the SPF paradigm describes only the movement of an agent, we applied a set of simple heuristics for the remaining actions. The attack heuristics dictates that the agent attacks any agent which gets closer than half of the critical distance [10]. The mating heuristics encourages the mating of isolated SPF agents, but restricts the mating of SPF agents inside groups. As an emergent property, this heuristics leads to moderate size, relatively stable groups of SPF agents.

3.11 CxBRAgent: Context Based Reasoning

Context-based Reasoning (CxBR) is a paradigm intended to model human tactical behaviors [5]. Contexts encapsulate knowledge about appropriate actions needed to address specific situations. The CxBR paradigm is composed of a tactical agent, mission context, major contexts, sub-contexts and sentinel rules which control the transitions between contexts.

`CxBRAgent` was implemented using eight different context constructs: 1) The ExploreContext is the default context of the mission. 2) The BackTrackContext

is called from ExploreContext when there is nothing new to explore in the map at the current location of the agent. The agent will then retrace its step and search for new places to explore. 3) The AttackContext is deployed when there is a hostile entity within the sensor range. 4) The AvoidContext represents the case when there is a hostile entity within the sensor range and the agent cannot attack the other agent. The agent will move away from the hostile agent, trying to avoid being chased or attacked by the other agent. 5) The EatContext is called from either the ExploreContext or BackTrackContext when there is food within the sensor range. The agent will move towards the food and invoke the eat command on the food resource. 6) The FleeContext is invoked when the agent have been attacked. The agent attempts to flee away from an attacker. 7) The MateContext is invoked rules when the agent can mate. 8) The NearDeathContext is invoked when the energy level of the agent is below a certain threshold. It will attempt to extend its lifeline by spawning another agent.

The CxBR agent implements a simple path-planning algorithm which allows it to navigate an internal representation of the global FFM map. The same path-planner is used when approaching objects in the agent's local sensor range.

3.12 KillerAgent: Simple Heuristics

The `KillerAgent` is implemented using a set of simple heuristics. The agent keeps track of direction, failed moves and successful moves. If the number of consecutive failed moves exceed a predefined threshold then a direction switch is performed. The same simple idea is used if too many moves have been successful. However, the threshold is higher in this case. When the agent detects food in its sensor it will immediately navigate to it and eat the food. If the agent senses other agents within its sensor range it will prioritize an attack over any other action. If the agent itself is attacked it will flee. This agent does not utilize the multiply feature or any teamwork strategies.

4 Implementation Effort

An important consideration in the choice of an agent paradigm is the effort and complexity of the implementation. Everything else being equal, a simpler implementation is frequently preferred, as it leads to a more maintainable code, with usually smaller number of defects.

The software engineering metrics for each implementation are shown in Table 3. The total lines of code (LOC) in conjunction with total lines of code inside all method bodies (MLOC) are an indicator of the development effort for each paradigm. In addition, we use cyclomatic complexity [12], to measure the complexity of the conditional flow within each implementation. We show values for the maximum cyclomatic complexity (MCC) found in a single method as well as the sum of cyclomatic complexity (SCC) over the complete implementation of the agent. All these measurements were applied strictly to the specific agent code and did not include the services provided by the environment neither external libraries. The only paradigm using an external library was NeuralLearner.

Table 3 shows that the NeuralLearner required more developer effort than any other paradigm. Also, the NeuralLearner was implemented using an external neural network library not included in the software metric calculations. The behavioral models such as social potential fields and crowd modeling have a relatively low count of lines, while agents with explicit programming models, such as CBR, CxBR or RuleBasedAgent have a relatively large number of code. As expected, KillerAgent is the most trivial agent, implemented with only 96 lines of code where 75 lines of code reside in method bodies.

Table 3. Canonical software metrics for each paradigm. LOC = Lines of code, MLOC = Method lines of code, MCC = Max cyclomatic complexity, SCC = Sum of cyclomatic complexity. Paradigms marked with (*) have external libraries that are not included in the metric calculations.

Name	LOC	MLOC	MCC	SCC	SCC/MLOC
AffectiveAgent	223	117	31	48	0.41
GenProgAgent	647	477	25	143	0.30
Reinforcer	313	236	13	52	0.22
CBRAgent	1357	1060	31	320	0.30
RuleBasedAgent	706	536	18	176	0.33
NaïveAgent	327	289	81	118	0.40
GamerAgent	1259	944	24	343	0.36
CrowdAgent	425	345	13	87	0.25
NeuralLearner*	1454	1119	36	297	0.27
SPFAgent	229	179	48	55	0.30
CxBRAgent	1135	689	21	298	0.43
KillerAgent	92	75	17	20	0.27

We use the SCC to MLOC ratio for comparison of cyclomatic complexity between the paradigms. AffectiveAgent, NaiveAgent and CxBRAgent all have ratios greater than 0.4. This means that for every line of code (inside method bodies) there is on average 0.4 conditional flow statements. These values indicate high complexity in learning and decision making within the paradigms. In contrast, Reinforcer, CrowdAgent and KillerAgent have low SCC to MLOC rations ranging from 0.22 to 0.27.

5 Findings

5.1 Development Process

The process of developing the twelve agents for this comparative study was organized in two stages. In the first, closed phase, the developers were working on the agents in isolation. Only a very simple random agent and the initial version of the KillerAgent was provided as an illustration of the API. The testing in

this phase was performed mostly in non-competitive settings. Some developers have tested their agents in competitive scenarios, by running their agents under different type strings. In some learning based paradigms the developers created "bootstrap" agents as training partners or ways to collect training information.

In the second, open phase of the development process, regular competitive runs were organized, which allowed the developers to observe the behavior of their own and of competing agents. However, the developers could not use opponent agents in scripted training scenarios. The developers had the opportunity to further develop or fine tune their agents to improve their performance in these competitive runs. We encouraged the developers to share their heuristics with each other, and emphasized that the goal is not to obtain the agent with the highest performance, but to best express the "spirit" of the paradigm. Nevertheless, a certain level of competitive pressure did develop between the developers.

By allowing encounters between the agents during the development process, we tried to guarantee that the agent can not win the game by a "surprise tactic" which might have been overlooked by the other developers. An example of this occurred when one of the developers discovered a hole in the game API, through which the agent could change the type string during the game and thus prevent attacks by masquerading as a friendly agent. This security hole was patched.

The comparison study allowed us to observe the different processes through which the developers analyzed the problem, approached the implementation, debugged or refined the agents, and reacted to the initial performance results. We found that the choice of the paradigm was critical in determining the flow of the development process. Although the developers were encouraged to use disciplined software engineering approaches, these turned out to be feasible only in the paradigms which involved explicit programming.

For paradigms where the behavior of the agent was determined by learning or encoded in variables such as force fields or affective or grouping parameters, software engineering techniques could only be applied to the development of the framework of the implementation. The main problem was that the results of a certain chunk of development effort was difficult to predict. Re-starting the learning process with a new set of parameters had frequently led to an agent with a lower performance. Adjusting the parameters of the force field to achieve a new behavior frequently invalidated previously achieved behaviors. All this made it difficult to apply quality assurance techniques.

5.2 The Limits of Learning

All paradigms which relied on learning (Neural Networks, Genetic Programming and Reinforcement Learning) have been successful in creating agents which can survive in the environment in the absence of predators. For all paradigms the developers spent significant time designing learning scenarios in which the algorithms can be steered in the right direction. This was made difficult by the fact that these scenarios had to be populated with agents. The most problematic paradigm from this point of view turned out to be the neural network agent, whose supervised learning algorithm required an existing agent to perform the

scenario to generate "correct" input and output pairs. Admittedly, this difficulty would be irrelevant in applications where such an imitation target exists and can be used at will.

All agents were successful in learning policies and inclinations (such as the ideal value of aggression), and they performed better than human intuition for these parameters. However, the learning agents were unsuccessful in learning (essentially, discovering) algorithms. The farthest they got was discovering approaches for collision avoidance, speed control for approaching the food, or avoiding to get stuck. However, we were not successful in developing path planning algorithms (such as an approach for visiting food locations). The genetic programming approach was the only learning model which would represent (and, theoretically evolve) such a model. However, an examination of the evolved genetic programs showed that they were very far from evolving anything like a path planning algorithm. This inevitably put them at a disadvantage against explicitly programmed agents which can deploy advanced algorithms such as A*, path planning, approximate Hamiltonian cycles and incrementally built internal maps.

5.3 A Rose by Another Name

Many paradigms led to surprisingly similar implementations, while giving very different interpretations to the variables involved. For instance, the developers of affective models `AffectiveAgent` and `CrowdAgent` used the variables describing the emotional states as just another state variable and applying regular programming techniques on them, on hindsight labeling them with emotional significance (the write-up for affective agents contained terms such as "emotional quagmire" for being stuck in a local minima). On the other hand, developers of other agents tended to assign anthropomorphic significance to their state variables ("the agent gets angry"), even if their paradigm did not require it. A similar phenomena was observed related to contexts. Context based reasoning, as implemented in the `CxBRAgent` requires the developer to actively identify the context of the agents operation and describes ways to handle it. However, the concept of context was actively used in at least four other agents. The `SPFAgent` and the `CrowdAgent` ended up deploying very similar attraction and repulsion forces, starting from different physical models and very different high level interpretations.

5.4 The Importance of the Heuristics

Although the game was not easy (humans playing it at first time did not perform better than agents), human users could easily come up with rules of thumb which offered significant performance increase. **The ease of representing these heuristics in the agents was a determining factor in the performance**. Agents in which this could be done only in a very convoluted way (such as the learning agents, and, in lesser degree, the potential field, crowd and affective models), had scored the worst in direct comparisons, and led to significant frustration.

5.5 "Paradigm-Pure Models Considered Harmful" or "Let Us Now Praise the Paradigm"?

Finally, let us consider a very general question: are paradigms useful at all in agent development?

To answer this, we need to first clarify the difference between an agent development paradigm and an algorithm. A paradigm is a guiding principle around which the agent development is organized. Some of the paradigms compared in this paper are also general purpose algorithms, which can be used in limited parts of the agent, without requiring an overall commitment from the developers.

The question is whether there is any advantage of organizing the development of an agent around a paradigm. This study required paradigm-pure implementations from the developers, that is, the developers were not allowed to borrow elements from other paradigms. In general, academic research projects would more likely be insisting on paradigm purity, as opposed to product development projects in the industry. This is partially justified by the different deliverables of an academic research group vs. a development team in the software or hardware industry. If our object is the study of a certain paradigm, paradigm purity is a natural choice. But what about the case when the objective is to optimize some type of performance measure?

In our study, the developers' subjective opinion was strongly against the paradigm purity requirement. Many developers felt a significant peer pressure to add additional heuristics, at the expense of the paradigm, to correct perceived performance and behavior problems.

Certainly, some paradigms made it exceedingly difficult to transfer human knowledge and rules of thumb to the agents, leading to performance problems. But, the other side of the coin is that an unchecked freeform development leads to a random collection of heuristics whose interactions are poorly understood. A good example is the NaïveAgent whose heuristics provided good performance in competitive scenarios, but which starved itself of resources by exponential multiplication when it was alone on the map. Of course, this could have been corrected with another heuristic, but it still leaves the basic problem unsolved.

In general our study supports the choice of a paradigm which can provide a coherent narrative to the development process, but it is not so restrictive that it would hinder the transfer of human knowledge to the agent. In particular, the ability to incorporate pre-existing algorithms is critical to the development of high performance agents.

6 Conclusions

In this paper we report on the findings of a study in which twelve paradigms of agency were compared in an environment inspired from strategy games and artificial life. A more extensive report on the study, together with source code and playable simulation runs is available from the website `http://netmoc.cpe.ucf.edu/Yaes/index.html`.

The research reported in this paper was partially supported by a fellowship from the Interdisciplinary Information Science and Technology Laboratory (I2Lab) at the University of Central Florida.

References

1. A. Aamodt and E. Plaza. Case-based reasoning: foundational issues, methodological variations, and system approaches. *AI Commun.*, 7(1):39–59, 1994.
2. W. Banzhaf, P. Nordin, R. E. Keller, and F. D. Francone. Genetic programming - an introduction: On the automatic evolution of computer programs and its applications. In *Morgan Kauffman Publishers Inc.*, 1998.
3. L. Bölöni and D. Turgut. YAES - a modular simulator for mobile networks. In *8-th ACM/IEEE International Symposium on Modeling, Analysis and Simulation of Wireless and Mobile Systems MSWIM 2005*, 2005.
4. E. Bouvier, E. Cohen, and L. Najman. From crowd simulation to airbag deployment: Particle systems, a new paradigm of simulation. *J. Electronic Imaging*, 6(1):94–107, 1997.
5. A. J. Gonzalez and R. H. Ahlers. Context-based representation of intelligent behavior in simulated opponents. In *Proceedings of the Computer Generated Forces and Behavior Representation Conference*, 1996.
6. S. Hanks, M. E. Pollack, and P. R. Cohen. Benchmarks, testbeds, controlled experimentation, and the design of agent architectures. *AI Magazine*, 14(4):17–42, 1993.
7. Hodjat and Shahrzad. Introducing a dynamic problem solving scheme based on a learning algorithm in artificial life environemtns. In *IEEE International Conference on Neural Networks. IEEE World Congress on Computational Intelligence.*, pages 2333–2338, 1994.
8. J. H. Holland. Adaptation in natural and artificial systems. In *University of Michigan Press, Ann Arbor*, 1975.
9. J.V.Neumann and O.Morgenstern. *Theory of Games and Economic Behavior*. Princeton University Press, 1944.
10. J. R. Koza. Genetically breeding populations of computer programs to solve problems in artificial intelligence. In *Proceedings of the Second International Conference on Tools for AI, Herndon, Virginia, USA*, pages 819–827. IEEE Computer Society Press, Los Alamitos, CA, USA, 6-9 Nov. 1990.
11. M. Likhachev, M. Kaess, Z. Kira, and R. C. Arkin. Spatio-temporal case-based reasoning for efficient reactive robot navigation. 2005.
12. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2:308–320, 1976.
13. T. M. Mitchell. *Machine Learning*. WCB/McGraw-Hill, 1997.
14. J. Reif and H. Wang. Social potential fields: A distributed behavioral control for autonomous robots. In *Proceedings of the International Workshop on Algorithmic Foundations of Robotics (WAFR)*, pages 431–459, 1995.
15. D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. In *Parallel distributed processing: explorations in the microstructure of cognition, vol. 1: foundations*, pages 318–362. MIT Press, Cambridge, MA, USA, 1986.

16. R. C. Schank. *Dynamic Memory: A Theory of Reminding and Learning in Computers and People*. Cambridge University Press, New York, NY, USA, 1983.
17. M. Scheutz. Useful roles of emotions in artificial agents: A case study from artificial life. In D. L. McGuinness and G. Ferguson, editors, *AAAI*, pages 42–48. AAAI Press / The MIT Press, 2004.
18. L. Yaeger. Computational genetics, physiology, metabolism, neural systems, learning, vision and behavior or PolyWorld: Life in a new context. In C. G. Langton, editor, *Artificial Life III, Proceedings Volume XVII*, pages 263–298. Addison-Wesley, 1994.
19. G. N. Yannakakis, J. Levine, J. Hallam, and M. Papageorgiou. Performance, robustness and effort cost comparison of machine learning mechanisms in FlatLand. *IEEE Proceedings of the 11th Mediterranean Conference on Control and Automation*, June 2003.

Augmenting BDI Agents with Deliberative Planning Techniques

Andrzej Walczak, Lars Braubach, Alexander Pokahr, and Winfried Lamersdorf

Distributed Systems and Information Systems
Department of Informatics, University of Hamburg
D-22527 Hamburg, Germany

Abstract. Belief-Desire-Intention (BDI) agents are well suited for autonomous applications in dynamic environments. Their precompiled plan schemata contain the procedural knowledge of an agent and contribute to the performance. The agents generally are constrained to a fixed set of action patterns. Their choice depends on current goals, not on the future of the environment. Planning techniques can provide dynamic plans regarding the predicted state of the environment. We augment a BDI framework with a state-based planner for operational planning in domains where BDI is not well applicable. For this purpose, the requirements for the planner and for the coupling with a BDI system are investigated. An approach is introduced where a BDI system takes responsibility for plan monitoring and re-planning and the planner for the creation of plans. A fast state-based planner utilizing domain specific control knowledge retains the responsiveness of the system. In order to facilitate integration with BDI systems programmed in object-oriented languages, the *planning problem* is adopted into the BDI conceptual space with object-based domain models. The application of the hybrid system is illustrated using a propositional puzzle and a multi agent coordination scenario.

1 Introduction

BDI is a well established model of agency [1] based on the Theory of Practical Reasoning [2]. Early BDI-systems have been devised with the idea in mind to overcome the poor performance of propositional planners controlling agents in dynamic environments at that time. The systems are based on two central ideas. One of them is the reactive planning, comparable with hierarchical planning systems [3], the other is deliberation [4,5].

Planning, an approach central to Artificial Intelligence (AI) research, is substantial for rational agent behavior. It is a method that aids agents in solving complex problems in synthetic and natural environments. Although planning systems are devised for means-end reasoning and are capable to find actions that achieve goals, they are less useful to decide, which goals to pursue [6].

Due to advances in planning techniques and understanding of *planning problems*, it seams reasonable and interesting to combine the strength of flexible means-end reasoning given by deliberative planners with the timely reactivity

R.H. Bordini et al. (Eds.): ProMAS 2006, LNAI 4411, pp. 113–127, 2007.

and goal deliberation capabilities carried by BDI systems. It is also interesting to analyze suitability of planning approaches to BDI agents in real world applications.

In order to benefit from both paradigms one needs to consider the strengths of both paradigms. There are multiple ways to compose the systems and the outcome is different dependent on their properties and features. As stated above, hierarchical planning techniques are comparable with BDI. Their strength lies in the evaluation of future environmental states and a constructed proof that a course of action will achieve goals under the preconditions. On the other hand, BDI systems handle dynamic environments more efficiently and are capable of both: reactive behavior, and maintenance of long term goals. They sacrifice the optimality and correctness of their planning algorithms for performance.

Both paradigms deal with generation of actions and share common ideas, so there are concerns which parts of a control and planning problem will be delegated to a planner and which to the BDI subsystem. This determines the choice of the BDI component and the planning algorithm. The overall architecture depends strongly on those choices.

A hybrid system can be built twofold. The planner may be applied to produce long term plans and to hand over single parts to a reactive BDI subsystem for the execution. This approach invokes serious performance concerns especially in dynamic environments where continuous changes force the planner to re-plan - a process with performance penalties comparable to planning itself. Generally, planning algorithms have been devised for one shot planning and are well suited for a solution of a single planning problem. They are rather less useful to maintain long term intentions of an agent.

The other way round is to augment the BDI system with a relatively simple planner that is invoked from the BDI controller and used for the purpose of creating short-term plans that need a proof of correctness. The last approach is used in this work to join the best from both paradigms.

The remainder of this paper proceeds as follows. In Section 2 we define the concept of a *planning problem* used for this work. Section 3 discusses the choice of a planning algorithm. In Section 4 we propose a way to integrate a planning component into a BDI framework. Section 5 presents two application examples of the hybrid system. Related work is presented in Section 6 and a conclusion is given in Section 7.

2 Planning Concepts

The basis for planning is given in the form of a *planning problem*. In order to represent a planning problem one needs at least to describe states of the world and how these states may change due to agent's actions. In a restricted classical view, this can be given by a model of a state-transition system $\Sigma = (S, \mathbb{A}, \gamma)$ where S is the set of states, A is the set of actions and $\gamma : S \times \mathbb{A} \rightarrow S \cup \{\perp\}$ is the transition function mapping a state and action to another state. $\perp$ is the illegal state being a result of a not applicable action. The planning problem is

given by a triple $\mathcal{P} = (\Sigma, s_0, g)$ where $s_0 \in S$ is the initial state and g is the description of a goal state inducing the set $S_g := \{s \in S \mid s \text{ satisfies } g\}$ [7].

The following definition of a planning problem has been found useful for the purpose of this work. It deviates from a standard definition by introducing utility functions interpreted as agent desires D. The other difference is an object-based representation of states build upon the sets O, A and V. The initial state s_0 has been understood as the current state s_c to reflect the fact that the planning takes place at runtime.

Definition 1. *A* state *is a tuple* $s =< G, \sigma, s_p, a_s >$ *where:*

- G *is a stack of agent goals in that state and each goal is a difference function* $g_{\in G} : S \to \mathbb{R}$ *revealing an approximate distance from the state to the goal in a common weighted measure.*
- $\sigma : O \times A \to V \cup O$ *is a partial function assigning values to the attributes of objects.* O, A *and* V *are taken from the model of a planning problem described below.*
- s_p *is the parent state of this one.*
- a_s *is the action applied to* s_p *state yielding* s.

The planner reasons about states of the environment and the agent's mental states. The environment is represented by object-based models given by an assignment function σ over a set of objects O and attributes A given below. For short-term planning only the immediate goals are interesting for the planner. A stack of goals G allows for ordered hierarchical decomposition. It is assumed that goals at this level of planning have been filtered through the deliberative BDI process and are consistent with each other.

Definition 2. *A* planning problem *is a tuple* $P =< \mathbb{A}, s_c, D, O, A, V >$ *where:*

- $\mathbb{A}$ *is a set of all actions available to the agent.*
- s_c *is the current state of the environment.*
- D *contains agent desires, in respect to possible solutions, assumed not to change within the short scope of operational planning.*
- O *contains the objects from the planning domain with attributes from the set* A *taking values from the set* V.

Desires are inverse utility functions $d_{\in D} : S \to \mathbb{R}$ on the states and reflect (not necessarily coherent) mental attitudes of the agent. Both desires and goals influence actions chosen by the agent, but only goals represent concrete points in the future state space, that the agent has to achieve. They also have direct impact on the choice of future goals.

Each action $a \in \mathbb{A}$ is a tuple $a =< p_a, \gamma_a, \omega_a >$ with p_a being a predicate over a state telling if the action is applicable. γ_a and ω_a are transition functions over the set of goals and attribute assignment functions respectively. If action a is applicable to s, the application transforms it to a new child state $s' =< \gamma_a(G), \omega_a(\sigma), s, a >$. This yields a new set of goals $G' = \gamma_a(G)$ and new attribute assignment function $\sigma' = \omega_a(\sigma)$.

3 Planning Algorithm

Planning has been assumed to be a higher cognitive activity than reacting and controlling behavior and has been granted more computational resources. We introduce a planner at a level below BDI control and deliberative behaviors. Given such an architectural design, the choice of a planning algorithm is restricted to a particular subset. With an advanced BDI system, one is equipped with reasoning and a strong conceptual framework, so there is no need to duplicate the functionality of both. To guide our choice of planning algorithms, the latest results from planning competitions [8] have been used. Planners entering such competitions have been tested on many standardized planning examples. Better performance indicates the right choice of an algorithm. Changing demands of successive planning competitions favored approaches that are easily expandable to new planning concepts.

In order for the system to remain responsive to circumstances that induced the planning task, the planner needs to operate under tight timing constraints. This fact emphasizes performance, not the generality or cognitive adequacy of a planning algorithm. State-based planners augmented with domain specific knowledge have been shown superior to partial-order planners, in that respect [8]. They are also easily applicable to many planning domains including propositional, numeric, timed, continuous and contingent domains.

The following planning algorithm is a state-based search algorithm working on an agenda of states. The main loop examines states from the agenda and expands them searching through the state space for one that achieves some or all of the goals. It terminates, when the agenda gets empty or when the time limit intended for planning is exceeded (cf. line 4). Please recall that states are tuples of the form $< G, \sigma, s_p, a_s >$. Each state has an assigned inverse utility estimate through the function $e : S \rightarrow \mathbb{R}$.

```
PLAN(𝔸, s_c, D, T)
 1  best ← s_c
 2  e(best) ← ∞
 3  agenda ← {s_c}
 4  while |agenda| > 0 and t_c < T
 5  do s ← remove_best(agenda)
 6      if e(s) < e(best) and |G_s| <= |G_best|
 7        then best ← s
 8      Options ← generateOptions(s, 𝔸)
 9      for  each {a =< p_a, γ_a, ω_a > | a ∈ Options}
10      do G' ← γ_a(G)
11          s' ←< G', ω_a(σ), s, a >
12          removeSatisfiedGoals(s')
13          e(s') ← inverseUtility(s', D) + goalsDistance(s', G')
14          insert(s', agenda)
15  return best
```

For many real world problems it is impossible for an agent to enumerate all action instances. Even in discrete domains, the number of possible actions becomes prohibitive. In a concrete design one would delegate the task of generating a set of applicable actions to a separate *option generator* component. Based on a set of action schemata and the actual situation it generates all applicable actions (cf. line 8). Lines 10 and 11 apply the transition function. In line 12 all satisfied goals are removed from the top of the goal stack of the new state. In order to sort states by their utility, a new estimate is calculated in line 13 before the state is inserted into the agenda. The variable implementation of *pop* and *insert* methods allows for different state exploration strategies.

There are two basic types of actions. *Concrete actions* modify object models in a state by changing the σ assignment function. In this case, the goal manipulation function is an identity function $\gamma_a = id$. The other type of actions are abstract ones called *decomposition methods* that remove a goal from the top of the goal stack G and replace it by a list of totally ordered subgoals. Such an action has generally no effect on the models of the environment.

The domain specific knowledge used to guide the planner is hidden in the action applicability predicates p_a, in the goal distance functions $g \in G$, and in the inverse utility functions $d \in D$ (cf. sec. 5). Including a reference to a parent in a state allows for complex temporal conditions over the course of actions, like safety and liveness ones [9]. The estimate is computed using the heuristic function below:

$$e(s) = \sum_{d \in D} d(s) + \sum_{g \in G} g(s)$$

This choice of this heuristic embodies the judgments that optimal solutions are anyway computationally expensive.

On success, the planning algorithm returns a state where every immediate goal posed to the planner is satisfied. When the algorithm fails to find a complete plan, the best plan in respect to the estimate and the number of unachieved goals is returned.

The emphasis has been put on the performance and use of domain specific knowledge. The difference to other linear planning algorithms based on the state-space search is the explicit representation of agent intentional structure manipulated during the process of planning. E.g. the STRIPS planner reasoned with a goal stack, this planner reasons *about* a goal stack in each planning step. This makes the planner similar to a state-based HTN planner with ordered decomposition. The understanding of agent's desires as heuristic and utility functions used to guide the planning process is also specific to this work.

4 Integration with a BDI Framework

A generic BDI agent architecture (based on [10,4]) is illustrated in Figure 1. One of the central components is the goal hierarchy housing higher level goals including desires and important events. The goals may depend on each other in

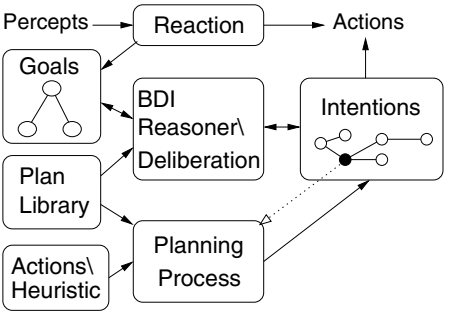


Fig. 1. Generic BDI architecture with a planner. Planning is an activity of the agent (filled circle) and produces new intentions.

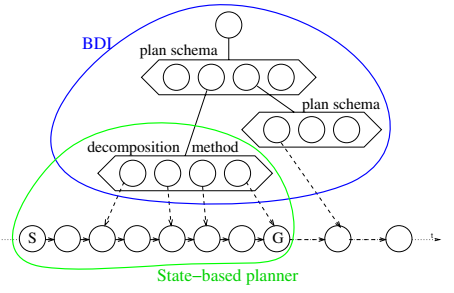


Fig. 2. Schematic illustration of a plan in the hybrid system

a forest like structure. This component includes the upper part of the intentional structure of an agent and provides it with motivation and reason for action.

Goals are used by the BDI reasoner to chose among plan schemata and to create the intentional structure. In object-oriented BDI frameworks plan schemata stored in a plan library are defined by classes of plans. The intentional structure is given by current plan instances. Another common component of the architecture is a reactive subsystem. Triggered by belief changes or a percept, it generates new goals for the reasoner. At the meta-level a goal deliberation process may influence the reasoning and execution. The goal deliberation component analyses dependencies among goals and modifies the intentional structure accordingly to agent's preferences.

4.1 Preparing a Planning Problem Instance

For a specific goal the BDI reasoner may use the planner to create a dynamic plan. This is done, whenever all existing plan schemata have failed or there is an explicit preference for the planner specified in the agent description file(cf. sec. 5). First an instance of a *planning problem* must be created. This requires mapping of agent goals and beliefs to the specific planner representation. All planner goals represent a distance measure defined over states with object-based models described above. There is no BDI system, known to the authors, that define goals as a distance measure with respect to different attribute types and dimensions. The domain designer performs the mapping. This manual approach utilizes the knowledge of the designer, who may provide adequate weights for attribute dimensions and distance functions for goals that cannot be represented in the object-attribute-value form.

In the planning process, the states are stored in a tree like database, that may consist of a large number of states, each storing a subset of beliefs. For planning being efficient in space and time only the relevant beliefs need to be

stored in a state. The choice of action schemata determines, which beliefs will be changed, and which will remain unaffected by the process. Object instances must be decomposed into their attributes in order to avoid copying solid objects including not relevant attributes. In the prototype, this process is performed manually, due to the complexity.

For example, in the loader dock scenario (cf. sec. 5) the position of a worker and the packet being held by the worker change. They are affected by agent actions and must be reflected in the state representation. On the other hand, a domain time model stores movements of other workers and their different positions with respect to time. The model is not affected by any actions and may be accessed directly form the agent's belief-base. In the blocks' world example, the relative position of a block is relevant for the planning process, but its color remains unregarded.

The actions work directly on the state representation. The central point of an action is the application method, including the applicability predicate and transition functions for the stack of goals and object descriptions (cf. sec. 5). This method summerizes all domain knowledge needed for planning with respect to the action. A wide range of conditions and types of search control knowledge may be specified this way. They are represented in the programming language of choice. The same is true in respect to the effects of an action. It is possible to derive this knowledge from already existing plan schemata of BDI systems, but for concrete examples studied, it showed not to be sufficient to aid the planner in an effective way. The actions are implemented in our approach by the application designer. Heuristics reflect the desires of an agent regarding created plans. They must be devised and implemented for the particular planning domain.

4.2 Planning and Execution

Prepared problem instances are handed to the planner and executed in a planning process like other agent's plans. The planner delivers created plans directly into the intentional structure and does not store them in a plan library. If plans generated by the planner were general enough in their nature, the designer of agent's knowledge could also precompile such plans in advance. Because plans are created at low-level, their parameters are tightly bound and they are applicable only to a particular situation. Storing such plans in the plan library would clutter it with instances used only once.

The resulting intentional structure may be seen in Figure 2. The upper fragment contains a partially expanded branch of the BDI goal structure. One of the goals has been assigned to the planner. It starts in the current state S and uses a decomposition method to place intermediate points in the search space as subgoals on the goal stack. The subgoals are removed from left to right as they are achieved. The created plan is placed in the intentional structure for execution and has a BDI goal as the parent node. The lower sequence represents the envisioned sequence of states, which should be attained at the execution time successively.

In a dynamic environment, conditions change in an unexpected way. Monitoring is an activity testing for the correct execution of a plan extending into the future. This activity is controlled by the BDI system. In order to prove that the remainder of a plan is correct it needs to be checked against the current situation. A component similar to the planner is used to evaluate the remainder in a simulated environment given by the planning domain. The simulator is a greedy planner with the option generator replaced by an iterator over the plan's tail.

If the simulation or the execution of a plan step reports an error, the plan is abandoned like a BDI plan instance. The goal responsible for invoking the planner is still located in the intentional structure of the BDI agent. The BDI reasoner may mark the goal with failure and abandon it or if the goal was marked to retry plans, the planner will be asked again for a new plan. In this respect, the planning process may be seen as a single agent plan schema or as a presumably infinite set of plan instances. The choice is taken at the design time and marked using BDI properties on agent goals specified in the agent description. For example, in the loader dock the `PickupPacket` goal is excluded on a plan failure (cf. sec. 5).

4.3 Managing Plan Failures

BDI execution engines have been devised under the assumption that the number of plans for a goal is limited to a small number. Given the capability of a planner to create an infinite number of plan instances, the BDI reasoning engine would repeatedly instruct the planner to create plans even if there are no plans that would achieve the goal. On the other hand, as the situation changes the planner may find a plan in the future. There are four cases that need a decision on the part of a BDI reasoner:

I. The planner cannot find any way to improve the agent's situation. In this case, it returns an empty plan with no actions. The BDI reasoner may wait a specific time and retry finding a plan. The BDI goal is marked with a BDI flag carrying the delay time between failed and new planning process.
II. The planner could not find a correct plan satisfying all goals and subgoals. Following this plan would allow the agent to reach a state where the goals are satisfied, but it could also lead it into a dead-end if the plan contains irreversible decisions. On the other hand, the plan may be executed with the hope that future planning, starting in a better situation, will find a complete plan. The description of planning problem given to the planner, should include a flag specifying if incomplete plans are allowed as a result.
III. The planner could not find a complete plan, because all of the time designated for planning has been used up. The time limit is specified in the description of the planning problem given to the planner. If a plan cannot be found because the problem exceed the planning horizon of an agent, an incomplete plan will be returned that fits specified problem best. This case can be handled the same way like case II.

IV. A number of correct plan instances is returned in successive trials but they fail to reach the goal. In this case the domain description is too abstract and lacks the knowledge needed by the planner to recognize specific reasons for failure. For example, the speed of a robot depends on the load carried, but domain designer specified constant speed. Such failures are seldom, but must be accounted for in this design. When plans fail because of a more demanding setting than the one stated in the domain description, a counter on the goal for failed plans may be the most simple solution.

BDI systems with elaborate goal representation including failure and retrial semantics [11], already offer the provisions to handle the cases at goal and plan levels. The description of goals and the semantics of the reasoner may be extended to provide BDI flags described above. When goals are merely events processed by the system this task can be delegated to an additional controller component wrapping around the planner.

5 Examples

The planner presented in this paper has been implemented in the JAVA™ language [12]. To evaluate it with a BDI framework it was integrated with JADEX [13] – a BDI reasoning framework developed at the University of Hamburg. JADEX incorporates many ideas from BDI-systems, like PRS [10] or JACK™ [14] and provides new unique facilities to deal with goal deliberation.

Two domains have been designed and implemented to demonstrate the dynamics and reasoning capabilities of the hybrid system. In the standard blocks-world example a simple but though propositional domain is used for testing. The Loader Dock example contains a continuous and highly dynamic domain demanding real-time performance.

5.1 Blocks' World

The blocks-world domain is a standard testing domain for planners. The problem consists of a bunch of blocks that must be moved into a final configuration. It is one of the first problems investigated with planners and from the beginning it has been a challenge as the problem is clearly exponential with respect to the number of blocks used.

The control knowledge is borrowed from the $\mathcal{TAL}$PLANNER [9] and adapted to JAVA™. The desire of an agent is to keep the number of moved blocks low. The distance to a goal is the number of blocks in *bad towers* (i.e. towers of blocks not conforming with the target state).

Figure 3 shows an implementation of the control knowledge in JAVA™. Here all the applicability predicate and transition functions are arranged together in a single method. This way, the code is kept in one place and much of redundant computation is abandoned.

For example, the control knowledge of `PutToTable` demonstrates the use of a temporal condition. Whenever in the foregoing plan a block was placed already

```
public boolean applyTo(State state)  {
   Block block=(Block)state.get(LOAD);
   if (block==null) return false;

   State previous=state.getPrevious();
   while(previous!=null) {
      if (previous.get(DOWN, block)==null) return false;
      previous = previous.getPrevious();
   }
   state.set(DOWN, block, null);
   state.set(LOAD, null);
   return true;
}
```

Fig. 3. Control knowledge for the `PutToTable` operator. A temporal condition assures that a block is put to the table only once.

on the table, the action is no more applicable to the state considered. This assertion prevents blocks from being placed on table again and again.

Using global search with an agenda of 10, it requires about 10 seconds on an i586 400MHz machine to stack 100 blocks. These results are not surprising as the planner has been build upon reliable and approved planning techniques. It compares well with the state of art planners (cf. [8]).

5.2 Loader Dock

In the loader dock several workers wander around or carry packets between incoming trucks and the shelves. Their job is to unload packets from a full truck, or to deliver packets to an empty one, whenever trucks arrive at the store. The dock itself contains shelves where packets can be placed temporally. The shelves are separated by corridor ways, which are used by workers to transport the packets (cf. fig. 4).

This particular domain is fully observable in respect to a certain update interval. This update of mutual beliefs that is performed by the store agent. It is almost deterministic because agents cannot be sure if created plans will be executed in time as they predicted. The environment changes because of other agents and processes. This includes trucks coming in and going out at various time intervals. Most quantities, like the number of packets, trucks, workers and places, are finite, but attributes of domain objects like speed, direction, arrival and departure time are real valued. There are many processes and agents acting in this domain concurrently. Agents do have common goals to handle the job at the storehouse, but share resources like time and corridor space. The storehouse depicted in Figure 4 is modeled using a discrete grid of points connected with each other through pathways. Workers and packets can take any position in the store and may head towards any real valued direction.

Planning takes very little time for this small domain. The approximate time of path planning on an i686 3GHz machine is from 1ms to 10ms. Because way

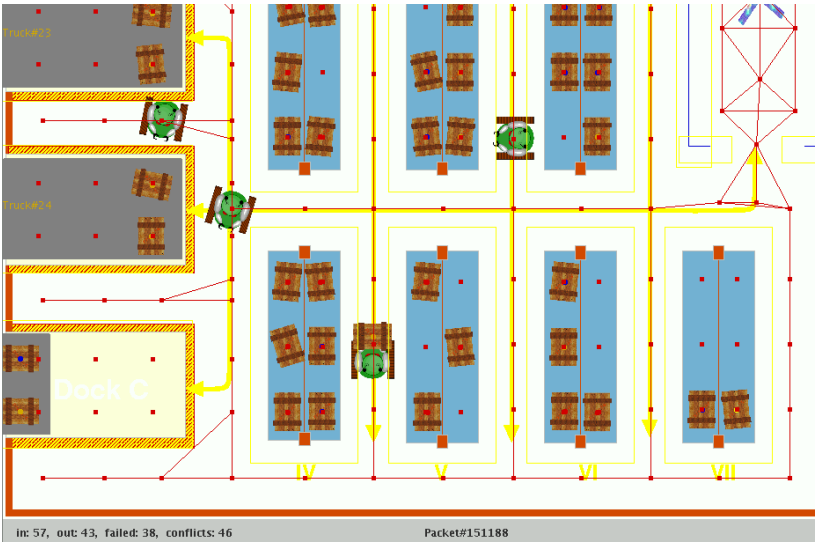


Fig. 4. The graphical interface of the loader dock example

points in the grid are static, all movement actions are precomputed in advance, which speeds the planning process.

When a worker makes an intention to move somewhere, it communicates this intention to other ones for the purpose of coordination. A trajectory of the movement, described by points and exact time-values, is sent to each worker, so it can update its beliefs about future changes in the environment. This information is stored in a `domain time model` representing the beliefs concerning prospective workers' positions. This information guides the planning process of each worker, so none of them intersects a path of another.

Goals to load or unload truck packets are distributed using the FIPA Contract Net Interaction Protocol [15]. Such a goal contains packet identification and a deadline. The worker responding with the fastest plan schema wins and commits to the execution. Other workers are informed about this intention. Figure 5 demonstrates a pickup goal. It represents a condition, where the agent is carrying a packet. `exclude="when_failed"` requests the BDI reasoner not to start the planner again after the initially created plan fails. The deliberation section tells the reasoner to suspend pursuing other goals with types: `Go, PickupPacketC, PutdownPacketC`. It prevents an agent from trying to go in different directions and to respond to calls for proposals, while processing this goal. The goal represents the BDI sphere of responsibility. It contains the provisions, as mentioned earlier, for re-planning (specified with flags), deliberation and monitoring (using the target condition).

A corresponding planning domain is specified in Figure 6. The planning process takes place in an instance of `PickupPlan` triggered by a `PickupPacket` goal. This part of the agent description file (ADF) specifies the responsibility of the

```
<achievegoal name="PickupPacket" exclude="when_failed">
   <parameter name="packet" class="Packet"/>
   <deliberation cardinality="1">
      <inhibits ref="Go"/>
      <inhibits ref="PickupPacketC"/>
      <inhibits ref="PutdownPacketC"/>
   </deliberation>
   <targetcondition>$beliefbase.packet!=null</targetcondition>
</achievegoal>
```

Fig. 5. Specification of the `PickupPacket` goal

```
<plan name="pickup" search="global" agenda="200" time="2000">
   <body>new PickupPlan()</body>
   <trigger>
      <goal ref="PickupPacket"/>
   </trigger>
   <heuristic>new TimeHeuristic()</heuristic>
   <operator>new MoveOp($to)</operator>
   <operator>new PickupOp()</operator>
</plan>
```

Fig. 6. Specification of the `Pickup` plan

planning component. Searching for the best plan to pickup a packet is done using a global search strategy. In order to confine the planning horizon the planning agenda is limited to 200 planning alternatives and the time given to the planner is 2000 ms. By default the planner should return correct plans only. The heuristic used expresses agent desire for plans taking less time to execute. The operator `MoveOp` changes the position of a worker and the operator `PickupOp` picks up the packet specified in the goal.

The use of a planner at the operational level helped worker agents to plan their activities and coordinate among themselves by communicating their intentions. They were successful to plan in a dynamic cooperative scenario with discrete and continuous resources. The BDI part has been effectively applied to control the dynamics of execution and has kept constraints over goals satisfied. The hybrid system has remained reactive and was able to adapt to different load factors.

6 Related Research

Architectures with strong emphasis on artificial intelligence include a planning component as their central part. An example system designed with BDI and planning in mind is InterRRaP [16]. It includes a local planning layer utilizing a hierarchical planner. The representation of procedures and goals follows that of a hierarchical task network (HTN) planner and has a declarative form. In the layered architecture, the planner takes control over a reactive subsystem.

The topic of joining procedural BDI reasoning engines with decision theoretic planners is not new. System have been built that proved especially useful in domains featuring enough time for planning, like in the example of PROPICE-PLAN [17] featuring a blast furnace domain. PROPICE-PLAN extends the dMars system with a state based planner IPP. CYPRESS is a system composed of the hierarchical planner SIPE-2 upon the Procedural Reasoning System (PRS) [10] also following a layered approach. Both systems are glued together with The ACT Formalism [18].

In the work of De Silva [19] a hierarchical planner JSHOP [20], is used to produce plans for the Jack™BDI system. Domain descriptions for the planner are created from the BDI plan schemata at compile time. The approach limits the programmer to a subset of possible programming solutions given by the intersection of the planning language and the BDI language and their possible transformations. It is also unclear, how to generalize this approach to a wider class of BDI systems.

BDI representation of agent internal mental states states can be mapped to the STRIPS [21] notation forth and back [22]. This has been done on an abstract BDI interpreter called X-BDI [23] and augmented with GRAPHPLAN. The mapping is a structure transformation of beliefs, desires and intentions into a propositional notation that is used by the planner so beliefs and actions are constrained by the propositional STRIPS domain representation.

These approaches aimed at technical or theoretical feasibility. There was no concern about generality or performance. In our opinion, they are not well applicable to planning of low-level control tasks. GRAPHPLAN and IPP are generic state-based planners using generic heuristics. The range of problems solved by these planners is limited and they are overpowered by planners applying domain specific knowledge [8]. They require to state the planning problem declaratively as a set of propositions. Object-based domain modeling approaches fit better with newer BDI frameworks designed in object-oriented languages such as JACK™ [14] or JADEX [13].

SIPE-2 and INTERRRAP planning layer use hierarchical decomposition in the space of partially ordered plans. The use of this planning space gives more degrees of freedom to the planner. The choice of such an algorithm has to be carefully elaborated. It is generally not the question, how to give the agent all the possible options, but how to restrict the choice to a minimal subset that needs to be considered. In fact, this is one of the advantages of BDI systems that constrain the options to a small number precompiled plan schemata. BDI systems also use deliberation and filtering techniques to further decimate the choices.

7 Conclusion

We have investigated the composition of a deliberative planner with a BDI framework forming a new hybrid system with combined characteristics. The use of a state-based planner on a planning problem extended by BDI concepts space

allows to easily merge those two paradigms. Implementing the planner in an object-oriented language and representing the planning domain with object-based models further facilitates the integration with BDI frameworks devised with such concepts in mind. The main requirement for the planner was performance at the low-level of execution. The planning problem representation included many places for application of domain specific knowledge including action preconditions, goal distance functions and utility functions.

Further an integration schema was proposed, where the BDI system is used as system controller responsible for the upper part of the intentional structure. It uses a deliberative planner in situations where precompiled plans are hard to devise in advance. In this schema the planner performs short term planning and produces plans with a constructed proof of correctness. The resulting plans are handed directly to the scheduler. Four cases have been identified that require changes in the semantics of goal handling in the BDI framework. These situation may trigger a re-planning process in order to create plans that better suit the problem on hand.

The approach has been verified in a puzzle domain to test the scalability of the planner showing comparable results with existing planning systems. The whole hybrid system has been investigated on the basis of the loader dock scenario. Here multiple agents had to plan their way through a packet store and coordinate their activities in order to cope with the load introduced by trucks coming in and going out. Worker agents could handle this domain because their intentional structure has been completed by plans created at runtime. On the other hand, the BDI reasoning could retain its reactive and deliberative characteristic.

The future of this work includes advances on the part of the planner. Techniques should be investigated to include concurrent planning and planning under uncertainty and embed it into a BDI reasoning framework. We anticipate that development of complex planning domains would require modeling and debugging tools. The abstraction given by knowledge representation is particularly important to this planning approach. Techniques for the representation of planning domains should be used to facilitate it.

References

1. Georgeff, M.P., Pell, B., Pollack, M., Tambe, M., Wooldrige, M.: The belief-desire-intention model of agency. In: Intelligent Agents, 5th International Workshop, ATAL'98. Springer, Paris (1998) 1–10
2. Bratman, M.E.: Intention, Plans, and Practical Reason. Harvard University Press, Cambrige, MA (1987)
3. De Silva, L., Padgham, L.: A comparison of BDI based real-time reasoning and HTN based planning. In: AI 2004: Advances in Artificial Intelligence, 17th Australian Joint Conference on Artificial Intelligence, Cairns, Australia, Springer (2004) 1167–1173
4. Bratman, M.E., Israel, D.J., Pollack, M.E.: Plans and resource-bounded practical reasoning. Computational Intelligence **4** (1988) 349–355

5. Pokahr, A., Braubach, L., Lamersdorf, W.: A goal deliberation strategy for BDI agent systems. In: Third German conference on Multi-Agent System TEchnologieS (MATES-2005). (2005)
6. Shut, M., Wooldridge, M.: The control of reasoning in resource-bounded agents. The Knowledge Engineering Review **16**(3) (2001)
7. Ghallab, M., Nau, D., P.Traverso: Automated Planning: Theory and Practice. Morgan Kaufmann Publishers (2004)
8. Edelkamp, S., Hoffmann, J., Littman, M., Younes, H.: The 4th international planning competition 2004 (IPC-2004) (2004) Hosted at the International Conference on Automated Planning and Scheduling 2004 (ICAPS-2004).
9. Kvarnström, J., Magnusson, M.: TALplanner in IPC-2002: Extensions and control rules. Journal of Artificial Intelligence Research (JAIR) **20** (2003) 343–377
10. Georgeff, M.P., Lansky, A.L.: Reactive reasoning and planning: An experiment with a mobile robot. In: Proceedings of the sixth National Conference on Artificial Intelligence (AAAI-87), Seattle, Washington (1987) 677–682
11. Braubach, L., Pokahr, A., Moldt, D., Lamersdorf, W.: Goal representation for BDI agent systems. In: The Second International Workshop on Programming Multi Agent Systems. (2004) 9–20
12. Walczak, A.: Planning and the belief-desire-intention model of agency. Master's thesis, University of Hamburg (2005)
13. Pokahr, A., Braubach, L., Lamersdorf, W.: Jadex: A BDI reasoning engine. In Bordini, R., Dastani, M., Dix, J., Seghrouchni, A., eds.: Multi-Agent Programming, Kluwer Academic Publishers (2005)
14. Busetta, P., Ronnquist, R., Hodgson, A., Lucas, A.: JACK intelligent agent - components for intelligent agents in Java (1999)
15. FIPA: FIPA Contract Net Interaction Protocol Specification. FIPA. (2001)
16. Fischer, K., Müller, J.P., Pischel, M.: Unifying control in a layered agent architecture. Technical Report TM-94-05, Deutsches Forschungszentrum für Künstliche Intelligenz GmbH, Kaiserslautern, DE (1994)
17. Despouys, O., Ingrand, F.F.: Propice-plan: Toward a unified framework for planning and execution. In Biundo, S., Fox, M., eds.: Recent Advances in AI Planning, 5th European Conference on Planning, ECP'99, Durham, UK, Springer (1999) 278–293
18. Wilkins, D.E., Myers, K.L., Wesley, L.P.: Cypress: Planning and reacting under uncertainity. In Burstein, M.H., ed.: ARPA/Rome Laboratory Planning and Scheduling Initiative Workshop Proceedings. Morgan Kaufmann Publishers Inc., San Mateo, CA (1994)
19. De Silva, L., Padgham, L.: Planning on demand in BDI systems. In: International Conference on Automated Planning and Scheduling, Monterey, California (2005)
20. Nau, D.S., Au, T.C., Ilghami, O., Kuter, U., Murdock, W., Wu, D., Yaman, F.: SHOP2: An HTN planning system. Journal of Artificial Intelligence Research **20** (2003) 379–404
21. Fikes, R.E., Nilsson, N.J.: STRIPS: a new approach to the application of theorem proving to problem solving. Artificial Intelligence **2**(3–4) (1971) 189–208
22. Meneguzzi, F.R., Zorzo, A.F., da Costa Móra, M.: Propositional planning in BDI agents. In: Proceedings of the 2004 ACM symposium on Applied computing, ACM Press (2004) 58–63
23. Móra, M.C., Lopes, J.G., Viccari, R.M., Coelho, H.: BDI models and systems: Reducing the gap. In: Proceedings of the 5th International Workshop on Intelligent Agents, Springer (1999)

ALBA: A Generic Library for Programming Mobile Agents with Prolog

Benjamin Devèze, Caroline Chopinaud, and Patrick Taillibert

Thales Airborne Systems
2 avenue Gay-Lussac, 78851 Elancourt - France
{firstname.lastname}@fr.thalesgroup.com

Abstract. This paper presents ALBA, a generic library dedicated to the commissioning of mobile agents written in Prolog. This library offers a handful of mechanisms for autonomous agent creation, execution, communication and mobility, whose implementation strongly respects the principles of robustness, decentralization of data, flexibility and genericity. In this perspective, the following paper mainly focuses on ALBA architecture and implementation with an emphasis on the technical choices which were made to provide these essential features. It therefore presents an innovative migration protocol, a research algorithm of agents solely identified by their names. It exposes some considerations about communication handling in a fully decentralized environment and some ideas towards a distributed modularity of systems. It also highlights an agent model, called Reasoning Threads, that is being used on top of ALBA to program cognitive agents.

1 Introduction

Since the emergence of multiagent systems (MAS), the corresponding research community has grown considerably and has been hard at work to provide agent communication standards mainly based on the speech act theory [2]. Important efforts have been made to formalize the main characteristics of agents, focusing on agent-oriented languages able to describe the behaviour of an intelligent agent. An abundant literature can be found about MAS related concepts like social attitudes, organization, cooperation or autonomy.

Unfortunately, the design of practical tools that can effectively support MAS programming and deployment appears to miss the necessary maturity to be widely adopted and used for large-scale industrial applications. A remaining gap persists between theories and concrete implementations that prevents us from taking the full benefits of this technology.

In order to demonstrate the added-value of the multiagent paradigm and to convince the remaining sceptic researchers and industrials, it is necessary to provide efficient programming constructs that facilitate the implementation of the essential concepts used in MAS. It is now admitted that MAS deal with flexibility, robustness, decentralization, modularity and scalability [21]. This should be kept in mind when developing new tools in this domain, so as not to alter these valuable qualities.

R.H. Bordini et al. (Eds.): ProMAS 2006, LNAI 4411, pp. 129–148, 2007.

Despite the numerous approaches and platforms architectures that have been proposed for agents commissioning, there is no general agreement on a particular method that would combine all the advantages of the agent paradigm. Platforms are often too centralized and often rely on imperative object-oriented languages, like Java, for agents implementation, which are not so well suited for this task [18]. It is especially the case for the kind of applications we have in mind in our group, that could be characterized as an attempt to apply the multiagent methodology to what is generally called real-time applications built on multitasked operating systems. These applications, which in our case concern mission systems embedded in aircrafts (sea or ground surveillance, coordinated observation missions by UAV -Unmanned Air Vehicles-, etc.) are generally complex since they not only manage a lot of tasks simultaneously but also rely upon complex algorithms whose duration cannot always be predicted (Artificial Intelligence approaches are more and more often necessary to implement the requirements of the new mission systems in preparation). To explore the various possible ways to make these systems evolve from a multitask to a multiagent perspective, a powerful implementation language capable of rapid agent model experimentation and AI algorithms development was needed. The most simple infrastructure was also needed in order to be able to easily merge our agents in an existing system and prove, without a complete redesign, that the multiagent approach was an alternative to present practices. That is the reason why ALBA has been designed as a library rather than a platform as is most often the case. Mobility was also a point since it makes it really easier to commission our agents on changing environments (all agents can be created on one computer -whose access is easier or devoted to our experiments- and then dynamically moved to the available computers at the time of the experiment).

Section 2 exposes the main reasons that led to the development of ALBA. Section 3 gives a general overview of the main aspects of our system that, in a way, put it apart from the majority of other tools. In section 4 we go thoroughly into some practical considerations about communications handling and in section 5 a dynamic agent search algorithm is detailed. Section 6 explains in depth the migration protocol and offers some views about agents mobility. Section 7 introduces a specific agent model, called Reasoning Threads, that is being used on top of ALBA to program cognitive agents illustrating a possible usage of the library. Section 8 describes some industrial applications already implemented using ALBA and the Reasoning Threads. Finally, sections 9 and 10 draw the main lessons of our proposals and discuss related and future works.

2 Why a New Platform?

Recent years have seen a considerable growth in the number of platforms, with a current total of over 100 products. It is then legitimate to ask why it has been necessary for us to develop a new one. The first exigence we had was that the platform had to allow the commissionning of agents written in Prolog.

2.1 Why Prolog?

Without exhaustively listing all the qualities of Prolog, the main reasons that naturally led us to use it to implement ALBA and our agents are stated here.

First of all, thanks to its two main mechanisms of unification and resolution and thanks to its efficiency in manipulating tree structures, Prolog is very well suited to deal with artificial intelligence problems and has already proved it in the past. As a declarative language benefiting from the first-order logic expressiveness, it seems to be the perfect candidate to serve as a basis for new agent-oriented programming languages. Moreover, Prolog allows to dynamically modify source code and offers a good environment to implement introspection capabilities. It is also a good choice for incremental verification of systems which are constructed with provability in mind.

Another essential argument, in our concern, was the natural efficiency of Prolog. It permits to develop and test very quickly some new prototypes and ideas. Its inherent productivity constitutes a great benefit in research activities without affecting at all the readability or the maintainability of source codes.

Prolog is an interpreted language and so as with Java, the same source code can run on various platforms and operating systems which is important to fulfill portability requirements at minimum cost.

Though, it can be argued that the Prolog language is not well equipped to deal with some specific tasks like real-time processing, modern graphical user interface development or efficient implementation of naturally imperative algorithms. Solutions can be found using the bidirectional interfaces to C, C++, Java which are provided with most mature Prolog implementations.

Last but not least, these implementations come with all the functionalities needed for the system to work: TCP/IP sockets, processes handling, Input/Output, etc. Finally, they provide advanced debuggers, efficient garbage collectors, constraint solvers and all the facilities programmers can expect nowadays.

2.2 Related Works

Obviously this part is mainly focused on platforms based on or providing logic programming facilities.

QU Prolog [6] and Ciao Prolog [14] both are Prolog extensions which offer multithreading and multi-machine execution of Prolog code. Agent behaviour programming is done thanks to production rules but other models can be implemented. Both offer also a blackboard for memory sharing or synchronization.

Jinni [24] is a platform allowing the programming of agents in BinProlog and Java. Jinni is based on a simple Things, Places, Agents ontology. Things are Prolog term, Places are processes running on computers and Agents are collections of threads executing a set of goals. The threads and the agents can communicate by using a blackboard and term unification. The threads can migrate between Places to communicate with the other threads and particularly to accelerate the resolution. Jinni can be used, for example, to simulate stock market, with the blackboard allowing agents coordination. Jinni is an interesting platform to

program Prolog agents but the blackboard oriented communication is a limitation we prefered to avoid in our context.

Eel [7] also deserves a mention since it implements communicating processes with an original point to point communication through term unification. But asynchronous processes were looked for, as carried out in ALBA.

tuProlog [9] might have been a good candidate since its design enforces interaction which is essential when agent implementation is concerned. Its close integration to Java is also an interesting feature, let alone for programming man-machine interfaces. The TuCSoN architecture [8] which was developped from tuProlog is a good example of what is looking for with ALBA: a programming environment facilitating the implementation of various agent or coordination models adapted to our needs such as the coordination artifact for time-aware agents presented in [11].

Thus, it looks as if several opportunities were offered for Prolog agent commissioning. So, why an industrial as Thales chose exploring new tracks rather than exploiting the existing solutions ? One of the reason was that not all features we had in mind were gathered in a unique platform (mobility, decentralized agent search) but the main point was our need for a robust Prolog basis such as the one offered by SICS with Sicstus Prolog and the existence in the company of a lot of legacy code for artifical intelligence tools (interval constraint propagation, for example) or applications that simply could not be neglected just for the sake of agent programming.

When non-Prolog platforms implementing mobility are concerned, they generally rely on imperative object-oriented languages for agents implementation (and not on Prolog) and are often dedicated to one specific or a limited set of agent models which was not satisfying. To our knowledge, all mobile agent platforms offer a centralization (from a server providing agents management in a same context or machine, to a central server managing agents in different computers). In every case, the migration, the agent creation, the communication are done through a dedicated entity which knows the local agents and its remote equivalents. These principles mainly exist for scalability and security reasons because the platforms are often used in the Web context. Our MAS approach is quite different. Our main objective is to distribute the agents of a given MAS over several computers in order to reduce and adapt the system workload throughout the execution. Moreover, robustness and functioning simplicity are essential. So, we tried to distribute platform specifications and services into the agent through the ALBA library.

3 Overview of ALBA

3.1 Main Features

ALBA is a Prolog library dedicated to the commissioning of agents written in Prolog. It uses SICStus Prolog [1] which is a mature and complete Prolog implementation with high performance and industrial qualities.

ALBA offers the basic functionalities expected from a multiagent platform. It brings the necessary mechanisms for agents creation, execution, communication and mobility.

The first noticeable point about ALBA is its complete *decentralization*. It means that the code implementing the platform functionalities is embedded in each agent. In this perspective, neither any central program nor any kind of data centralization are required for it to work. Hence, as it has already been mentioned, ALBA can better be described as a predicate library rather than as a platform. Of course, decentralization raises a lot of problems, especially related to communications handling. A substantial part of this paper is devoted to the practical proposals we have implemented to tackle these issues.

ALBA is also about *genericity*. That means that no assumption is made on the agent models used. Therefore, ALBA can be used with any kind of agent models (Agent-0[22], AgentSpeak[25], BDI[19], etc.). Since, at this stage, the research community has not agreed on a specific universal model which can be used for any kind of applications, and since it is even doubtful that such a model exists, it seems to be the best way to proceed when industrial applications are concerned. Moreover, this approach greatly facilitates experimentations on various models and on the way they can be combined to reach our expected goals. In the same range of ideas, no assumption is made on the language used by the agents to communicate.

Another point of interest is *flexibility*. As a generic low-level tool, the library assures, purposely, a restricted range of basic functionalities. It is a core tool that can be extended at will to provide higher level functionalities, as mentionned in sections 7 and 8.

3.2 General Overview

An ALBA agent is constituted of two parts, the embedded library in charge of all the basic services (messaging, contacts, etc.) and the behaviour of the agent itself that is coded in Prolog. Each agent is an independent Prolog process and has a unique name that fully identifies it in the system. Therefore, in all the paper, the agents will be represented as in Figure 1. All agents can communicate via asynchronous messages.

ALBA has been developed in a multi-machine perspective and, of course, our systems can be distributed on a pool of computers over a network. For the remaining of the paper, the term computer will refer to any computer in a network where an *ALBA daemon* is running. These daemons are in charge of executing on remote machines, the creation or migration functions called by an

Fig. 1. An ALBA Agent

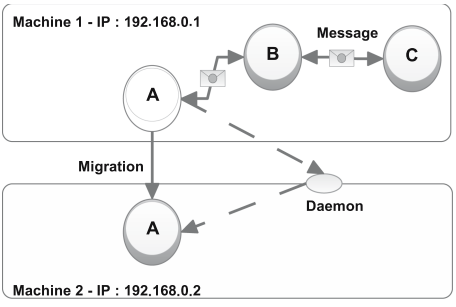


Fig. 2. ALBA Overview

agent and the associated data transfers (see section 6 for more details). Therefore, agents can be created locally or on remote computers and can migrate from a computer to another. A general overview is presented in Figure 2.

3.3 Towards Distributed Modularity

In ALBA, agents are created from *proto-agents* using the predicate:
`create_agent(+Host, +Name, ?Param, +Contacts, +ProtoAgent, -FullName)`.[1]

Proto-agents are to agents what classes are to objects in the object-oriented paradigm. Each agent running in a MAS can be viewed as a specific instance of a proto-agent. For the sake of reusability, proto-agents aim to be as generic as possible. They consist of the Prolog source code describing the "to-be-instantiated" agents behaviour and of any resource files they could need. Each proto-agent is stored as a directory or as an archive file and takes its name from its corresponding directory name or filename. Each newly created agent is given a specific workspace initialized from the content of the proto-agent it is based on. When an agent creates another agent from a proto-agent A, ALBA automatically searches A following the order given in its *proto-agent path*, querying remote daemons when necessary. The proto-agent is then automatically retrieved to localhost as a compressed archive and can be used to launch the new agent.

To accomplish their tasks, agents may need specific libraries, for example libraries dedicated to image analysis, interval computations, etc. Including these generic libraries in each proto-agent would be costly and nearly unmaintainable, that is why ALBA use quite the same mechanisms as for proto-agents to automatically find and retrieve required libraries which are shared at MAS level. To do so, ALBA offers the following predicates which are encapsulations of theirwell-known Prolog homologues: `alba_consult(+LibName)`, `alba_compile(+LibName)`, `alba_use_module(+LibName)`.

Now, let us suppose we want to run a MAS to simulate a mission with boats and planes using various libraries. We have nothing in our machine but we know

[1] A few predicates of the library are introduced but generally parameters are not described for being quite self-explicit.

the address of remote servers hosting required data. Provided we just have a daemon running on our computer, we can build a complete customized MAS using proto-agents and libraries coming from various machines, serving as distributed banks of generic agents and resources.

4 Communications Handling

Communications handling is a hard task that, at low-level, needs some knowledge about network protocols that seems far from intelligent agents problematics. However, communication is a fundamental aspect of multiagent paradigm, as being the only way for cognitive agents to share information. Indeed, it is by prohibiting the usage of complex sharing methods (shared memory, semaphores, etc.) that MAS can pretend to reduce the structural complexity of systems.

As stated before agents are able to communicate asynchronously through messages. In this view, ALBA offers direct point-to-point communications with `send_message(+Message, +Recipient)` predicate. Communications relied on TCP/IP sockets which can seem inappropriate for local communications but is required for remote transmissions.

Agents are identified by their names, whose uniqueness is ensured by ALBA using the following naming scheme, which only exploits information locally saved in each agent: *AgentName/SonName/etc.* Note that names are stored and manipulated as Prolog terms, allowing us to deduce immediately from an agent's name the identity of all his ancestors. Another interesting feature of this naming scheme is that it allows the merging of completely distinct MAS into a single one. Indeed, provided that the seed agents of each MAS to merge have a unique name -which is not a severe requirement-, it is clear that there won't be any name clash issues. Therefore, relying on the search method described in section 5, merging two distinct MAS can be done by simply linking one agent from each MAS to each other.

Exchanged messages are Prolog terms which is extremely convenient when it comes to parsing and analyzing their contents. No other assumptions are made about messages contents and, of course, every classical communication languages (KQML[16], FIPA ACL, etc.) can be used.

ALBA users can build up their systems on the following postulate: for the same pair of agents, messages ordering is preserved. More formally, if *m1* and *m2* are two messages sent from A to B, *m1* being transmitted before *m2*, then B will receive *m1* before *m2*. This is ensured by TCP protocol and the library internal mechanisms.

Error treatment is an important aspect of communications handling. ALBA acts as a layer on top of TCP/IP to manage every detail related to communications, such as connections, transmissions, proper disconnections and so on. It also has to deal with any potential low-level errors that may occur. Agent programmers work at a higher abstraction level and must not have to be preoccupied about these kind of considerations. Communications handling in ALBA, can be compared to ordinary postal service. It is, therefore, up to agents to prevent

any possible loss of essential information thanks to specific protocols. For example, ALBA comes with acknowledgments facilities, which can be used for synchronous communications if it becomes necessary.

It can be useful for an agent to use some appropriate messages treatment strategies. It becomes nearly inevitable for very solicited agents so as to rationally handle the mass of received messages. Strategies can give the precedence to some specific senders or to a given kind of messages that need to be processed in priority. That is why, in addition to the classical `read_message(-Message, ?Sender, +Timeout)` routine, ALBA provides the predicate `read_all_messages(-Messages, ?Senders)` that returns all the messages available at call time. It is possible to instantiate the variable *Senders* in order to get only the messages sent by given senders. Note that an agent messagebox is stored in memory using Prolog terms, allowing to easily handle advanced requests by unification.

5 Search Method

5.1 Introduction

All agents of the system are identified by their unique name. This is the only information accessible to the end user of ALBA. All communications being based on TCP/IP sockets, the library has to provide internal mechanisms to recover the IP address and port number of an agent from its name. In order to achieve this goal, we refused to use any kind of name servers or matchmaker agent (respectively white and yellow pages) or, more generally, to assign this task to any form of central system that would constitute potential drawbacks for our applications. Relying on a centralized approach would affect the robustness of ALBA because a single fault in this central organ could paralyse all the system. Moreover, excessive centralization constitutes a major bottleneck, as the central entity has to stay aware of every changes occurring in the MAS (agent creation, migration, etc.) and to answer all the queries of the agents willing to communicate. Hence, the central entity has to deal with a very important amount of messages with an overload risk. These issues can be partially solved using several matchmaker entities communicating with each other, ensuring the integrity of their names database and implementing mechanisms of data redundancy to prevent the system to fully depend on the existence of a single entity. This solution has, though, an important cost and that is the reason why an alternative way was chosen.

To address this problem[2], we aim at taking the best advantage of the natural distributivity of MAS and to exploit information stored locally in each agent. The problem can be viewed as a graph search problem where nodes represent agents and where arcs stand for connections between agents, i.e. an agent A is linked to an agent B if and only if A knows the correct IP address and port of B at a given time T. The problem is more complex than a traditional graph search because, here, the topology of the graph, since it is a model of the MAS, can

[2] So far, only the replacement of centralized "white pages" is achieved.

evolve dynamically during the search. It is also important to understand that, even if agent A has some information about B, we cannot be sure that these data are up-to-date.

Therefore, the search algorithm need to be able to successfully retrieve agents in an unstable graph whose arcs may be wrong, by propagating a wave of messages in the MAS. It needs to fulfill the three following objectives:

1. the wave of messages generated by the algorithm must come to an end, no matter what is the configuration of the MAS
2. if agent A searches agent B, which is in the same connected component, the algorithm must be able to find B
3. the amount of messages sent during the search need to be limited as much as possible

A Naive Algorithm. The general idea is very simple. When agent A wants to communicate with agent B and does not know how to reach it, A sends a search message to all its contacts. The search message contains: some necessary information to reach the search initiator (SI), a blacklist of already visited contacts (BL) and the name of the target agent (TA). Of course the algorithm works and fulfills the two first objectives but generates too many messages to be used. Indeed, in the worst case, i.e. N agents interconnected (complete graph) with a search for an agent absent from the MAS, it is straightforward to see that the algorithm induces a wave of $(N-1)!$ messages.

5.2 An Improved Algorithm

The idea, to limit the number of sent messages, is to exploit local information stored in each agent, i.e. its contacts, in order to anticipate a step forward. Thus, when an agent receives a search message it adds itself *and* all its non already visited contacts in the blacklist. Unfortunately, this algorithm does not ensure anymore that an agent will find any agent in its connected component. It is illustrated in Figure 3.

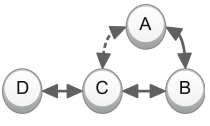


Fig. 3. Error in search

In this example, A wants to reach agent D. A sends a search message to its contacts B and C with A, B and C in the BL. A cannot effectively send the message to C because it has outdated localization information concerning C. Note that B holds updated position of C and could reach it, but of course it

does not even try to do so because C is in the BL. Even if D and A belong to the same connected component, A cannot find D anymore.

It is therefore necessary to add a local error treatment mechanism. An automatic ping pong procedure could have been used to ensure that each contacts are reachable before sending them the search message. It is not reliable and too heavy to be used in large MAS, especially when it appears that, in the majority of cases, there won't be any error to handle. That's why an alternative method has been proposed, described in algorithm 1, which we have called Waves-Search algorithm.

In this method, agents automatically take into account communication errors and resend their search messages with an appropriate corrected blacklist.

Algorithm 1. Waves-Search algorithm

```
1: if OwnName = Target Agent (TA) then                    ▷ I am the searched agent!
2:     Get in contact with the Search Initiator
3: else
4:     if TA ∈ OwnContacts and search message successfully sent to TA then
5:         return
6:     end if
7:     Forwards ← OwnContacts − Blacklist (BL)
8:     UpdatedBL ← BL + OwnName + Forwards
9:     for all Agent ∈ Forwards do
10:        Send updated message with UpdatedBL to Agent
11:    end for
12:    Errors ← List of agents in Forwards that could not receive the search message
13:    if Errors ≠ ∅ then
14:        NewForwards ← Forwards − Errors
15:        NewUpdatedBL ← UpdateBL − Errors
16:        for all Agent ∈ NewForwards do
17:            Send updated message with NewUpdatedBL to Agent
18:        end for
19:    end if
20: end if
```

This algorithm works well for applications involving a reasonable number of agents organized in favorable interconnected topologies. It is also important to understand that the algorithm is launched only a limited number of times to interconnect two agents at first or as an alternative procedure if an agent has lost some contacts. Though, it suffers some scalability issues and would not be suitable for massive MAS with all possible topologies. It can be viewed as a first step towards a fully effective dynamic search method in a decentralized environment.

The problem is very close to search processes in Gnutella-like unstructured and fully decentralized peer to peer networks and to classical application layer routing protocols, which are active fields of research [10]. A possible improvement would be to adapt Distributed Hash Table based methods like Chord, CAN, Pastry or

mobile ad hoc network routing protocols like DSR or AODV to the specificities of the problem. It may imply to soften the second objective offering only guarantees in probability to find an existing agent. Another very promising approach would be to exploit the agents genealogy, which can be deduced from agents name, in order to direct the search very quickly and with bound guarantees on the number of messages sent for all topologies.

6 Migration Protocol

Mobility, i.e. the support to the network transport of agent code and execution state, has become one of the fundamental feature any modern platform should provide. Mobile agent advantages, which are stressed in several papers, such as [4,17], explain this imperative requirement. Listing only a few of them, agent mobility allows network traffic reduction, dynamic MAS reconfiguration, load balancing and is of great support to improve scalability and fault-tolerance. This section describes the migration protocol used by ALBA and discusses its main characteristics.

6.1 Description

At any time, agents can use the predicate `migrate(+Host)`, to keep on with their work on any remote computer. Note that ALBA provides only the necessary mechanisms for agents mobility. Each agent chooses its target host and the best moment to migrate relying upon migration strategies established by the developer. This is of course reasonable considering that these strategies are application-dependant and stand at a higher abstraction level than ALBA. To achieve the migration task, ALBA proceeds as described in Figure 4.

1. The migrant wants to move to a remote host.
2. A clone of the migrant is created on the remote host.
3. The clone creates a connection with the migrant contacts, the migrant stops its activity and only forwards messages to its clone.
4. The connections between the migrant and its old contacts are cut.
5. The migrant process destroys itself, the clone has replaced it on the remote target.

Now that only a general overview of the migration protocol has been given, it is necessary to describe what happens in each agent playing a part in the procedure.

From the migrant perspective. The migrant first creates a clone of itself on the target computer. In practice, the workspace of the migrant is transferred by the ALBA layer in the migrant, to the remote ALBA daemon as a compressed archive file and the remote daemon launches the clone agent. The migrant reads all its pending messages and transfers all its messages and ALBA related internal

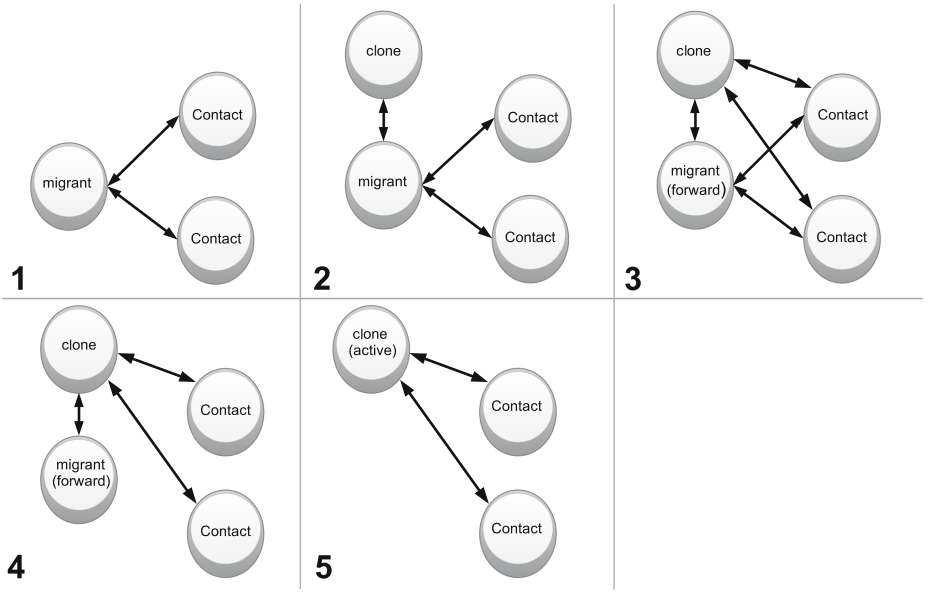


Fig. 4. Migration Protocol

data, such as its contacts, directly to its clone. All messages received during this phase are forwarded by the migrant to the clone. Note that all the forwarded messages are encapsulated properly to inform the clone of its real senders. Then, the migrant immediately sends an *end of migration* message to the clone and closes itself.

From the clone perspective. The clone is launched from the same source code as the migrant. It first connects to its father, i.e. the migrant, which is the only agent of the system aware of its existence. As described in section 4, its internal name has the form *migrant_name/clone(X)*. It then initializes its internal data with those provided by the migrant. As soon as it receives migrant contacts information, it sends them a special internal update message stating that it is the new agent named *migrant_name*. This message is automatically interpreted by the ALBA layer which just replaces migrant information with clone address and port. Upon the reception of the *end of migration* message from the migrant, the clone changes its internal name to *migrant_name* and calls the *restart* predicate that has to be written by the agent developer and define the first behaviour of the restarted agent.

From the migrant contacts perspective. From the migrant contacts everything is transparent. There is only a hidden substitution in their internal data from the migrant address and port to the clone address and port.

6.2 Discussion

One of the main interest of this protocol is that the migrant and its clone are running together during a very short period of time and that no messages are lost during this transitory phase. Indeed, if one of the migrant contact sends the migrant a message when it is still running, it will forward it to the clone. If the migrant is already closed but the contact has not already received the special internal update message, the message will be queued and sent as soon as the clone contacts it.

At first, it seems that ALBA provides only weak mobility because no migration of execution state is involved, the migrating agents are explicitly restarted at their destination. However, ALBA comes with the two following routines: `put_into_luggage(+Name, +Value)` and `get_from_luggage(+Name, ?Value)`, allowing to save and restore data in a specific part of the memory which is automatically transferred during a migration. These predicates are callable at any time during agent execution and represent a convenient way to manage what can be viewed as a *migration luggage*. The agent model described in section 7 can be fully defined by its internal data. Thus, using their migration luggage properly, as described in section 7, agents implemented with this model are able to completely resume their execution after a migration, which becomes a transparent procedure. Therefore, the migration strength depends of the agent model which is used, the library offers strong migration at agent level if the model used is *migration compliant*, i.e. if the agent behaviour can be resumed by the sole knowledge of its *luggage*.

7 Reasoning Threads: A Model of Agency

As a low-level library dedicated to the commissioning of agents, ALBA is well suited to serve as the core of various agent model experimentations. Using directly the communication routines offered by ALBA would, of course, be inadequate to implement intelligent agents involved in complex interaction and coordination processes. This section illustrates the genericity of ALBA and shows how it can be used with a specific model based on Reasoning Threads (RTs) to deploy these agents.

7.1 Basic Concepts

Autonomy is a central concept of agency. Following [12], we adopt an operational definition of autonomy stating that *an agent A is autonomous with regard to an agent B if and only if B cannot predict definitely A decisions*. An agent can decide not to process a message e.g. because it has more important goals to satisfy or because of workload. Therefore, agents need to be able to react accordingly if they do not receive expected answers and must foresee commitments breaking. Agents can not predict the exact behaviour of other agents, but they can delimitate classes of alternative behaviours that can be expected. As a consequence, agents plans need to be conditional over possible actions/reactions

of other agents. Thus, thinking agents in term of autonomous entities constitutes a way to improve fault tolerance at the source, dealing with loss of messages, death of other agents or machine overload. Another central concept to reduce the structural complexity of systems is the *limited dependency*, we assume that interactions with other agents and the environment only take place by exchanging messages, prohibiting memory or resource sharing without an agent mediation. These two principles influence a lot the agentification of systems and the implementation of agents.

Messages play a central role. They are the only information agents perceive about others activities. Therefore, it has been necessary to provide an architecture that makes messages analysis and handling easier. As an agent gets dynamically involved in many interaction processes with various agents it has also been necessary to provide some mechanisms to properly handle different contexts simultaneously. As a matter of fact, when a human receives his mail he generally roughly sorts it on the basis of the sender and content of each received item and then link each received letter with a previous context or create new contexts to handle further mail exchanges attached to new topics. In the same range of idea, threads in forum allow the sorting of messages according to their object and topic and can consequently handle multiple contexts simultaneously. These metaphores were of great influence for the RT approach.

7.2 Description

The RT library acts as a layer on top of ALBA encapsulating the low-level routines offered by the library which are not available anymore to the agent programmer (Cf. Fig. 5).

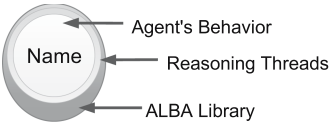


Fig. 5. An ALBA Agent based on RT

Each RT can be viewed as a context. A RT template is described as an extended finite state machine representing a procedural knowledge associated to a context. Thus, a RT conveys some explicit knowledge about the interaction processes taking place among agents in order to reach a goal in a specific context. When a RT template is instanciated, a local memory is created to store the data relevant to this context. A global memory is accessible to all RTs. All messages arrive to a *switch* which is in charge of the messages routing to the relevant RTs according to a set of grammar rules that work on the sender, the syntax and the content of incoming messages (Cf. Fig. 6). If a message can not be filtered by the switch, it is automatically directed to the default RT which plays a crucial role to identify new contexts and to handle unknown messages.

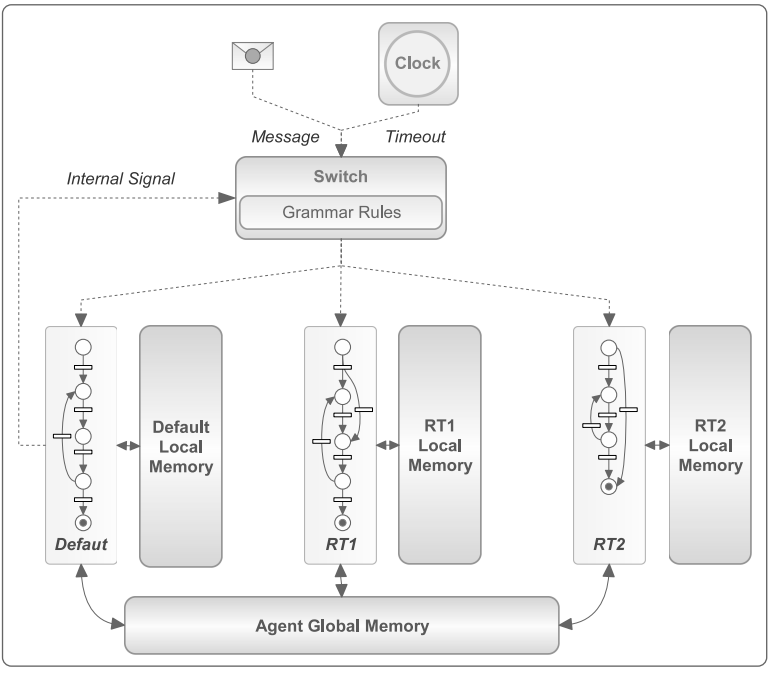


Fig. 6. Architecture Overview

RTs are dynamically instanciated and destroyed according to the evolution of the system. As an agent may be simultaneously involved in multiple interaction processes, multiple RTs can of course be instanciated at the same time. For example, if an agent answers to various proposals from different agents, it would possibly have multiple instances of RTs describing the Contract Net protocol [23] running at the same time to handle these interactions. Grammar rules, RT templates and currently instanciated RTs can be modified, added or deleted at run time allowing to dynamically adapt the behavior of running agents.

A RT consists of a set of states S_i, an initial state $begin \in S_i$, a final state $end \in S_i$, a set of state transition rules R_i, a local memory Mem_i and each state $s_i \in S_i$ is associated with a timeout $t_{s_i} \in T_i$ (in seconds or *off*) modifiable at runtime; that is $RT_i = \langle S_i, T_i, R_i, Mem_i \rangle$.

A transition rule consists of a condition and an associated action. The invocability conditions of a rule consist of the RT current state, an incoming event and a facultative filter that has access to local and global memories. An event can be (1) an incoming message from another agent (`msg(Message, Sender)`), (2) an internal signal (`signal(Signal, RT_ID)`) or (3) a timeout. Internal signals allow communications between various RTs which can be useful for example to resolve conflicts between various contexts. Each rule specifies an action that is executed when it is fired and the next state of the RT. An action can consist of any Prolog code including calls to encapsulated ALBA routines (message sending,

migration, agent creation) and calls to RT related routines (grammar rule modifications, RT modifications, RT invocations, memory modifications). Each rule is written as a Prolog clause:

```
rt(Type, State-in, State-out, Event, Action) :- Filter.
```

Agents plans may be incomplete or inaccurate and the knowledge to extend or correct them may become available only at runtime. Therefore, agents need to be able to extend and modify their existing plans and also to build new plans dynamically. For this reason, `State-out` can be a free variable in order to incrementally build a plan at runtime.

7.3 Execution Model

When the switch is initialized, the default RT and the agent global memory are created. The interpreter keeps up to date RTs related data like the names, current states, current timeouts and local memories of currently instanciated RTs. It also maintains a queue of pending events.

The execution cycle works as follows:

Algorithm 2. Execution Cycle

```
1: while there is an instanciated RT do
2:     compute the timeout to apply (minimum of the remaining timeouts of the currently instanciated RTs)
3:     read all pending messages during at most timeout seconds
4:     if there are messages to handle then
5:         apply the filtering_strategy routine to select the message m to process
6:         use the grammar rules on m to compute the RTs to trigger
7:     else
8:         wake up the RTs in timeout
9:     end if
10:    sequentially execute actions of the transitions that have been fired
11:    remove RTs in state "end"
12: end while
```

Since state transition rules are fired sequentially, it is not necessary to use mutual exclusion mechanisms to protect accesses to the global memory of the agent, memories will stay consistent while an RT action is executed. This approach reduces the systems complexity still allowing to handle multiple contexts simultaneously.

Note that the `filtering_strategy` predicate can be redefined to implement various strategies to give precedence to specific messages or senders according to the context. The default behaviour is a queue.

7.4 Mobility

As described in section 6, ALBA provides strong migration at agent level if the model used is *migration compliant* as is the case for the RT approach. Indeed,

when the migration routine is called in the body of an action, the interpreter ensures that the migration procedure is the last operation executed just after the branching to the new state, leading to the following steps:

1. the states, memories and related data are stored in the agent migration luggage
2. the migration predicate of ALBA is explicitly invoked
3. the revelant data are restored on target host when migration is achieved
4. the timeouts of currently instantiated RTs are updated
5. the execution is resumed

7.5 Discussion

Related works can be found in the field of coordination languages with COOL [3] or AgenTalk [15] which respectively introduced the notions of *conversations* and *scripts* which share the same philosophy as the RTs. These two languages do not use grammar rules for messages filtering but propose to use user maintained or automatically maintained identifiers to route incoming messages to conversations. The grammar rules of the RT approach are much more flexible since they can be modified at runtime and allow to route a given event to multiple running RTs. It also contributes to the clear definition of contexts based on the syntax and semantics of incoming events. Moreover, provided the introspection and metaprogramming facilities offered by Prolog and the language used to describe RTs, we defend the idea that it will be easier to extend the mechanisms introduced by COOL and AgenTalk to develop the flexibility and the introspection abilities of our agents.

A *script*, a *conversation* or a *RT* can be viewed as a procedural knowledge for an agent to reach a certain goal. In this perspective, these approaches have much in common with Procedural Reasoning System (PRS) [13]. However, PRS is mainly focused on a single goal-directed agent whereas the coordination languages are focused on social aspects describing protocols among agents. Therefore, we now aim at combining these approaches and extending the RT model with explicit goals and planification abilities in order to improve the proactivity of our agents which is so far limited to timeouts and internal signals processing. Thus, further works will naturally be focused on ways to achieve a good balance between goal-directed and reactive behavior in a timely fashion.

Looking at the execution cycle of the system, the question of actions duration is clearly of great importance to answer appropriately to incoming events in a timely fashion. In this perspective, [11] propose to delegate “long” computations to computational artifacts controlled by agents.

8 Applications

ALBA and the RT approach have already been used in various applications of our department. Only one of them is mentioned here: *Interloc*. Interloc is

a software for mobile marine targets localization. In Interloc, planes seek to detect boats while remaining stealth, i.e. without using their own radar, but by exploiting the targets emissions to deduce their positions. The system is implemented as a MAS. Each boat is represented by an agent, just as each plane. Another agent manages the graphical interface, another one makes measures and an agent by plane is in charge of localization computations. Therefore, a substantial number of agents (10 to 25) are to run and interact at the same time. Interloc was a perfect testbed application to validate ALBA and especially its migration protocol. Indeed, considering the significant number of agents running simultaneously, migration was interesting for load balancing purposes.

As explained, as a low-level tool, ALBA provides purposely a very limited set of functionalities. Thus, to carry out these applications, a lot of useful tools have been developed on top of it, which proved the flexibility and extensibility of the library. For instance, an agent was developed to facilitate human agents interactions with active MAS via a graphical interface. It mainly allows to create or kill agents and comes with a console to easily communicate with running agents by sending them messages.

ALBA has been well tested in practice and has proved to be efficient in achieving its tasks. The RT approach has proved to be extremely easy and convenient to use thanks to the natural separation of contexts which allows the programmer to focus on local problems attached to specific contexts. The ALBA library and its related tools are now mature enough to be used in larger scale industrial applications.

9 Future Works

Even though, ALBA is already quite functional, several aspects have to be improved and new functionalities need to be added.

The increasing development of agents mobility and the distribution of MAS over heterogeneous networks raised the question of security. Therefore, to be deployed in untrustworthy environments, ALBA needs to support cryptography mechanisms to provide communications encryption and agents authentication. In the same perspective, all manipulated archives need to be encrypted too.

In our applications, agents are to interact with entities that are not really agents but just some runtime devices providing a specific kind of function or service. There is, for example, an entity that is used by agents to display graphical interfaces. This is not a common agent, it does not use ALBA and cannot fully interact with other agents. Therefore, ALBA needs to properly handle this kind of entities introduced in [20], as a first-class abstraction in MAS under the name “artifacts”.

We have also begun to develop a generic library allowing to define events linked with any chosen predicates. Programmers can use these predefined events to inject their own code on specific points of interest. The library relies on Prolog introspection mechanisms. It would allow to easily customize ALBA with external files and without modifying its core. It could also be used to dynamically control agents as in [5]. More advanced studies are to be made to explore the potential benefits of this library.

10 Conclusion

We have presented in this paper a generic Prolog library called ALBA, dedicated to MAS deployment. We described thoroughly its architecture and implementation with an emphasis on the technical choices made to provide robustness, decentralization, flexibility and modularity. With a strong respect for these features, we introduced an innovative migration protocol, an agent research algorithm and some considerations about communications handling. We also highlighted some ideas to achieve a distributed modularity of agents. Relying upon the described mechanisms, it is already possible to merge completely distinct MAS, to tackle on-line repairing of agents or to stop any agent for some time and relaunch it later, minimizing the impact on the rest of the system. Part of our current work is focused on these experimentations, on ALBA improvements and on its applications.

Now that we have a usable library dedicated to MAS commissioning, our main concern is also to explore the best ways to express autonomous agents behaviour. A preliminary work has been presented in this paper with the description of an agent model based on Reasoning Threads. We now aim to extend this model and to propose a new declarative high-level agent-oriented programming language built on top of Prolog.

References

1. Available at `http://www.sics.se/isl/sicstuswww/site/index.html`.
2. John Langshaw Austin. How to Do Things with Words. *Clarendon Press*, 1962.
3. M. Barbuceanu and M. S. Fox. Cool: A language for describing coordination in multiagent systems. In Victor Lesser and Les Gasser, editors, *Proceedings of the First International Conference oil Multi-Agent Systems (ICMAS-95)*, pages 17–24, San Francisco, CA, USA, 1995. AAAI Press.
4. David Chess, Colin Harrison, and Aaron Kershenbaum. Mobile Agents: Are They a Good Idea? Technical report, IBM Research Division Report, 1995.
5. Caroline Chopinaud, Amal El Fallah Seghrouchni, and Patrick Taillibert. Prevention of Harmful Behaviors within Cognitive and Autonomous Agents. In *Proc. of the 17th European Conference on Artificial Intelligence (ECAI'06)*, pages 205–209, August 2006.
6. K. Clark, P.J. Robinson, and R. Hagen. Multithreading and message communication in Qu-prolog. *Theory and Practice of Logic Programming*, 1(3), 2001.
7. Torbjrn S. Dahl. The eel programming language and internal concurrency in logic agents. In *the Proceedings of the Workshop on Multi-Agent Systems in Logic Programming, (ICLP'99)*, Las Cruces, New Mexico, November 29 - December 4 1999.
8. Enrico Denti and Andrea Omicini. From tuple spaces to tuple centers. *Sci. Comput. Program.*, 41:277–294, 2001.
9. Enrico Denti, Andrea Omicini, and Alessandro Ricci. Multi-paradigm java-prolog integration in tuprolog. *Sci. Comput. Program.*, 57(2):217–250, 2005.
10. Gang Ding and Bharat K. Bhargava. Peer-to-peer file-sharing over mobile ad hoc networks. In *PerCom Workshops*, pages 104–108, 2004.

11. Cédric Dinont, Emmanuel Druon, Philippe Mathieu, and Patrick Taillibert. Artifacts for time-aware agents. In *Fifth Int. conf. on Autonomous Agents and Multiagents Systems (AAMAS 06)*, Hakodate, Japan, 8 - 12 May 2006.
12. Mark d'Inverno and Michael Luck. Understanding autonomous interaction. In *ECAI*, pages 529–533, 1996.
13. M. Georgeff and A. Lansky. Procedural knowledge. *Proceedings of the IEEE (Special Issue on Knowledge Representation)*, 74:1383–1398, 1986.
14. Daniel Cabeza Gras and Manuel V. Hermenegildo. The ciao module system: A new module system for prolog. *Electr. Notes Theor. Comput. Sci.*, 30(3), 1999.
15. Kazuhiro Kuwabara, Toru Ishida, and Nobuyasu Osato. Agentalk: Coordination protocol description for multiagent systems. In Victor Lesser, editor, *Proceedings of the First International Conference on Multi–Agent Systems*, page 455, San Francisco, CA, 1995. MIT Press.
16. Yannis Labrou and Tim Finin. A Proposal for a New KQML Specification. Technical Report TR CS-97-03, Computer Science and Electrical Engineering Department, University of Maryland Baltimore County, February 1997.
17. Danny B. Lange and Mitsuru Oshima. Seven Good Reasons for Mobile Agents. *Commun. ACM*, 42(3):88–89, 1999.
18. James Odell. Objects and Agents Compared. *Journal of object technology*, 1(1):41–53, 2002.
19. A. S. Rao and M. P. Georgeff. BDI-Agents: from Theory to Practice. In *Proceedings of the First Intl. Conference on Multiagent Systems*, San Francisco, 1995.
20. Alessandro Ricci, Mirko Viroli, and Andrea Omicini. Programming MAS with Artifacts. In *Proc. of the Third International Workshop on Programming Multi-Agent Systems'05*, pages 163–178, July 2005.
21. Pierre-Michel Ricordel and Yves Demazeau. From Analysis to Deployment: A Multi-agent Platform Survey. *LNCS*, 1972:93–105, 2001.
22. Yoav Shoham. Agent-oriented programming. *Artif. Intell.*, 60(1):51–92, 1993.
23. Reid G. Smith. The contract net protocol: High-level communication and control in a distributed problem solver. *IEEE Transactions on Computers*, 29(12), 1980.
24. Paul Tarau. Jinni: Intelligent mobile agent programming at the intersection of java and prolog. In *Proceedings of PAAM'99*, London, 1999.
25. D. Weerasooriya, A. Rao, and K. Ramamohanarao. Design of a Concurrent Agent-Oriented Language. In M. Wooldridge and N. R. Jennings, editors, *Intelligent Agents: Theories, Architectures, and Languages (LNAI Volume 890)*, pages 386–402. Springer-Verlag: Heidelberg, Germany, 1995.

Bridging Agent Theory and Object Orientation: Agent-Like Communication Among Objects

Matteo Baldoni[1], Guido Boella[1], and Leendert van der Torre[2]

[1] Dipartimento di Informatica. Università di Torino - Italy
{baldoni,guido}@di.unito.it
[2] University of Luxembourg
leendert@vandertorre.com

Abstract. This paper begins with the comparison of the message-sending mechanism, for communication among agents, and the method-invocation mechanism, for communication among objects. Then, we describe an extension of the method-invocation mechanism by introducing the notion of "sender" of a message, "state" of the interaction and "protocol" using the notion of "role", as it has been introduced in the `powerJava` extension of Java. The use of roles in communication is shown by means of an example of protocol.

1 Introduction

The major differences of the notion of agent w.r.t. the notion of object are often considered to be "autonomy" and "proactivity" [25]. Less attention has been devoted to the peculiarities of the *communication capabilities* of agents, which exchange messages while playing roles in protocols. For example, in the contract net protocol (CNP) an agent in the role of initiator starts by *asking for* bids, while agents playing the role of participants can *propose* bids which are either *accepted* or *rejected* by the Initiator.

The main features of communication among agents which emerge from the CNP example are the following:

1. The message identifies both its *sender* and its *receiver*. E.g., in FIPA the acceptance of a proposal is:
   ```
   (accept-proposal :sender i :receiver j :in-reply-to
       bid089 :content X :language FIPA-SL).
   ```
2. The interaction with each agent is associated to a *state* which evolves according to the messages that are exchanged. The meaning of the messages is influenced by the state. E.g., in the FIPA iterated contract net protocol, a "call for proposal" is a function of the previous calls for proposals, i.e., from the session.
3. Messages are produced according to some *protocol* (e.g., a call for proposal must be followed by a proposal or a reject).
4. The sender and the receiver play one of the *roles* specified in the protocol (e.g., initiator and participant in the contract net protocol).

R.H. Bordini et al. (Eds.): ProMAS 2006, LNAI 4411, pp. 149–164, 2007.

5. Communication is *asynchronous*: the response to a message does not necessarily follow it immediately. E.g., in the contract net protocol, a proposal must follow a call for proposal and it must arrive, no matter when, before a given deadline.
6. The receiver autonomously decides to comply with the message (e.g., making a proposal after a call for proposal).

The message metaphor has been originally used also for describing method calls among objects, but it is not fully exploited. In particular, message-exchange in the object oriented paradigm has the following features:

1. The message is sent to the receiver without any information concerning the sender.
2. There is no state of the interaction between sender and receiver.
3. The message is independent from the previous messages sent and received.
4. The sender and the receiver do not need to play any role in the message exchange.
5. The interaction is synchronous: an object waits for the result of a method invocation.
6. The receiver always executes the method invoked if it exists.

These two scenarios are rather different but we believe that the object-oriented (OO) paradigm can learn something from the agent-oriented world. The research question of this paper is thus: is it profitable to introduce in the OO paradigm concepts taken from agent communication? how can we introduce in the OO paradigm the way agents communicate? And as subquestions: which of the above properties can be imported and which cannot? How to translate the properties which can be imported in the OO paradigm? What do we learn in the agent-oriented world from this translation?

The methodology that we use in this paper is to map the properties of agent communication to an extension of Java, `powerJava` [3,4,5], which adds roles to objects. Roles are used to represent the sender of a message (also known as the "player of the role"), to represent the state of the interaction via role instances, allowing the definition of protocols and asynchronous communication as well as the representation of the different relations between objects.

The choice of the Java language is due to the fact that it is one of the prototypical OO programming languages; moreover, MAS systems are often implemented in Java and some agent programming languages are extensions of Java, e.g., see the Jade framework [8] or the JACK software tool [24]. In this way we can directly use complex interaction and roles offered by our extension of Java when building MAS systems or extending agent programming languages.

Furthermore, we believe that in order to contribute to the success of the Autonomous Agents and Multiagent Systems research, the theories and concepts developed in this area should be applicable also to more traditional views. It is a challenge for the agent community to apply its concepts outside strictly agent-based applications. The OO paradigm is central in Computer Science and, as observed and suggested also by Juan and Sterling [18], before AO can be widely

used in industry, its attractive theoretical properties must be first translated to simple, concrete constructs and mechanisms that are of similar granularity as objects.

The paper is organized as follows. In Section 2 we show which properties of agent communication can be mapped to objects. In Section 3 we introduce how we model interaction in `powerJava` and in Section 4 we discuss how to use roles in order to model complex forms of interaction between object inspired by agent interaction, we also illustrate the contract net protocol among objects using `powerJava`. Conclusions end the paper.

2 Communication Between Objects

When approaching an extension of a language or of a method, the first issue that should be answered is whether that extension brings along some advantages. In our specific case, the question can be rephrased as: Is it useful for the OO paradigm to introduce a notion of communication as developed in MAS? We argue that there are several acknowledged limitations in OO method invocation which could be overcome, thus realizing what we could call a "session-aware interaction".

First of all, objects exhibit only one state in all interactions with any other object. The methods always have the same meaning, independently of the identity or type of the object from which they are called.

Second, the operational interface of Abstract Data Types induces an *asymmetrical* semantic dependency of the callers of operations on the operation provider: the caller *takes the decision* on what operation to perform and it relies on the provider to carry out the operation. Moreover, method invocation does not allow to reach a minimum level of "control from the outside" of the participating objects [2].

Third, the state of the interaction is not maintained and methods always offer the same behavior to all callers under every circumstance. This limit could be circumvented by passing the caller as a further parameter to each method and by indexing, in each method, the possible callers.

Finally, even though asynchronous method calls can be simulated by using buffers, it is still necessary to keep track of the caller explicitly.

The above problems can be solved by using the way communication is managed between agents and defining it as a primitive of the language. By adopting agent-like communication, in fact, the properties presented in Section 1 – with the only exception of autonomy, (6), which is a property distinguishing agents from objects – can be rewritten as in the following:

1. When methods are invoked on an object also the object invoking the method (the "sender") must be specified.
2. The state of the interaction between two objects must be maintained.
3. In presence of state information, it is possible to implement interaction protocols because methods are enabled to adapt their behavior according to

the interaction that has occurred so far. So, for instance, a proposal method whose execution is not preceded by a call for proposals can detect this fact and raise an exception.
4. The object whose method is invoked and the object invoking the method play each one of the roles specified by the other, and they respect the *requirements* imposed on the roles. Intuitively, requirements are the capabilities that an object must have in order to be able to play the role.
5. The interaction can be asynchronous, thanks to the fact that the state of the interaction is maintained.

For a better intuition, let us consider as an example the case of a simple interaction schema which accounts for two objects. We expect the first object to wait for a "call for proposal" by the other object; afterwords, it will invoke the method "propose" on the caller. The idea is that the call for proposal can be performed by different callers and, depending on the caller, a different information (e.g. the information that it can understand) should be returned by the first object. More specifically, we can, then, imagine to have an object `a`, which exposes a method `cfp` and waits for other objects to invoke it. After such a call has been performed, the object `a` invokes a method `propose` on the caller. Let us suppose that two different objects, `b` and `c`, do invoke `cfp`. We desire the data returned by `a` to be different for the two callers.

Since we look at the agent paradigm the solution is to have two different interaction states, one for the interaction between `a` and `b` and one for the interaction between `a` and `c`. In our terminology, `b` and `c` interact with `a` in two distinct roles (or better, *role instances*) which have distinct states: thus it is possible to have distinct behaviors depending on the invoker. If the next move is to "accept" a proposal, then we must be able to associate the acceptance to the right proposal.

In order to implement these properties we use the notion of role introduced in the `powerJava` language in a different way with respect to how it has been designed for.

3 Modelling Interaction with `powerJava`

In [1,12,21,23] the concept of "role" has been proved extremely useful in programming languages for several reasons. These reasons range from dealing with the separation of concerns between the core behavior of an object and its interaction possibilities, reflecting the ontological structure of domains where roles are present, from modelling dynamic changes of behavior in a class to fostering coordination among components. In [3,4,5] the language `powerJava` is introduced: `powerJava` is an extension of the well-known Java language, which accounts for roles, defined within social entities like institutions, organizations, normative systems, or groups [7,14,26]. The name `powerJava` is due to the fact that the key feature of the proposed model is that institutions use roles to supply

the *powers* for acting (*empowerment*). In particular, three are the properties that characterize roles, according to the model of normative multiagent systems [9,10,11]:

Foundation: A (instance of) role must always be associated with an instance of the institution it belongs to (see Guarino and Welty [16]), besides being associated with an instance of its player.

Definitional dependence: The definition of the role must be given inside the definition of the institution it belongs to. This is a stronger version of the definitional dependence notion proposed by Masolo *et al.* [19], where the definition of a role must include the concept of the institution.

Institutional empowerment: The actions defined for the role in the definition of the institution have access to the state and actions of the institution and to the other roles' state and actions: they are powers.

Roles require to specify both *who can play the role* and *which powers are offered* by the institution in which the role is defined. The objects which can play the role might be of different classes, so that roles can be specified independently of the particular class playing the role. For example a role customer can be played both by a person and by an organization. Role specification is a sort of double face interface, which specifies both the methods required to a class playing the role (*requirements*, keyword "playedby") and the methods offered to objects playing the role (*powers* keyword "role"). An object, which plays a role, is empowered with new methods as specified by the interface.

To make an example, let us suppose to have a printer which supplies two different ways of accessing to it: one as a normal user, and the other as a superuser. Normal users can print their jobs and the number of printable pages is limited to a given maximum. Superusers can print any number of pages and can query for the total number of prints done so far. In order to be a user one must have an account which is printed on the pages. The role specification for the user is the following:

```
role User playedby AccountedPerson {
  int print(Job job);
  int getPrintedPages();
}

interface AccountedPerson {
  Login getLogin();
}
```

The superuser, instead:

```
role SuperUser playedby AccountedPerson {
  int print(Job job);
  int getTotalPrintedPages();
}
```

Requirements must be implemented by the objects which act as players.

```
class Person implements AccountedPerson {
  Login login;  // ...
  Login getLogin() {
    return login;
  }
}
```

Instead, powers are implemented in the class defining the institution in which the role itself is defined. To implement roles inside an institution we revise the notion of *Java inner class*, by introducing the new keyword `definerole` instead of `class` followed the name of the role definition that the class is implementing.

```
class Printer {
  final static int MAX_PAGES_PER_USER;
  private int totalPrintedPages = 0;

  private void print(Job job, Login login) {
    totalPrintedPages += job.getNumberPages();
    // performs printing
  }

  definerole User {
    int counter = 0;
    public int print(Job job) {
      if (counter > MAX_PAGES_USER)
        throws new IllegalPrintException();
      counter += job.getNumebrPages();
      Printer.this.print(job, that.getLogin());
      return counter;
    }
    public int getPrintedPages(){
      return counter;
    }
  }

  definerole SuperUser {
    public int print(Job job) {
      Printer.this.print(job, that.getLogin());
      return totalPrintedPages;
    }
    public int getTotalPrintedpages() {
      return totalPrintedPages;
    }
  }

}
```

Roles cannot be implemented in different ways in the same institution and we do not consider the possibility of extending role implementations (which is, instead, possible with inner classes), see [5] for a deeper discussion.

As a Java inner class, a role implementation has access to the private fields and methods of the outer class (in the above example the private method *print* of *Printer* used both in role *User* and in role *SuperUser*) and of the other roles defined in the outer class. This possibility does not disrupt the encapsulation principle since all roles of an institution are defined by who defines the institution itself. In other words, an object that has assumed a given role, by means of it, has access and can change the state of the corresponding institution and of the sibling roles. In this way, we realize the powers envisaged by our analysis of the notion of role.

The class implementing the role is instantiated by passing to the constructor an instance of an object satisfying the requirements. The behavior of a role instance depends on the player instance of the role, so in the method implementation the player instance can be retrieved via a new reserved keyword: `that`, which is used only in the role implementation. In the example the invocation of `that.getLogin()` as a parameter of the method `print`.

All the constructors of all roles have an implicit first parameter which must be passed as value the player of the role. The reason is that to construct a role we need both the institution the role belongs to (the object the construct `new` is invoked on) and the player of the role (the first implicit parameter). For this reason, the parameter has as its type the requirements of the role. A role instance is created by means of the construct `new` and by specifying the name of the "inner class" implementing the role which we want to instantiate. This is like it is done in Java for inner class instance creation. Differently than other objects, role instances do not exist by themselves and are always associated to their players.

Methods can be invoked from the players, given that the player is seen in its role. To do this, we introduce the new construct

$$receiver\ \texttt{<-(}role\texttt{)}\ \ sender$$

This operation allows the sender (player of the role) to use the powers given by "role" when it interacts with the receiver (institution) the role belongs to. It is similar to *role cast* as introduced in [3,4,5] but it stresses more strongly the interaction aspect of the two involved objects: the sender uses the role defined by the receiver for interacting with it. Let us see how to use this construct in our running example. The first instructions in the main create a printer object `hp8100` and two person objects, `chris` and `sergio`. `chris` is a normal user while `sergio` is a superuser. Indeed, instructions four and five define the roles of these two objects w.r.t. the created printer. The two users invoke method `print` on `hp8100`. They can do this because they have been empowered of printing by their roles. The act of printing is carried on by the private method `print`. Nevertheless, the two roles of `User` and `SuperUser` offer two different way to interact with it: `User` counts the printed pages and allows a user to print a job if the number of pages printed so far is less than a given maximum; `SuperUser` does not have such a limitation. Moreover, `SuperUser` is empowered also for viewing the total number of printed pages. Notice that the page counter is maintained

in the role state and persists through different calls to methods performed by a same sender/player towards the same receiver/institution as long as it plays the role.

```
class PrintingExample {
 public static void main(String[] args) {

   Printer hp8100 = new Printer();
   Person chris = new Person();
   Person sergio = new Person();

   hp8100.new User(chris);
   hp8100.new SuperUser(sergio);

   (hp8100 <-(User) chris).print(job1);
   (hp8100 <-(SuperUser) sergio).print(job2);
   (hp8100 <-(User) chris).print(job3);

   System.out.println("Chris has printed " +
     (hp8100 <-(User) chris).getPrintedPages() + " pages");
   System.out.println("The printer hp8100 has printed a total of " +
     (hp8100 <-(User) sergio).getTotalPrintedPages() + " pages");

 }
}
```

By maintaining a state, a role can be seen as realizing a *session-aware interaction*, in a way that is analogous to what done by cookies or Java sessions for JSP and Servlet. So in our example, it is possible to visualize the number of currently printed pages, as in the above example. Note that, when we talk about playing a role we always mean playing a role instance (or *qua individual* [19] or *role enacting agent* [13]) which maintains the properties of the role.

An object has different (or additional) properties when it plays a certain role, and it can perform new activities, as specified by the role definition. Moreover, a role represents a specific state which is different from the player's one, which can evolve with time by invoking methods on the roles. The relation between the object and the role must be transparent to the programmer: it is the object which has to maintain a reference to its roles. However, a role is not an independent object, it is a facet of the player.

Since an object can play multiple roles, the same method will have a different behavior, depending on the role which the object is playing when it is invoked. It is sufficient to specify which the role of a given object, we are referring to, is. In the example `chris` can become also `superuser` of `hp8100`, besides being a normal `user`

```
   hp8100.new SuperUser(chris);
   (hp8100 <-(SuperUser) chris).print(job4);
   (hp8100 <-(User) chris).print(job5);
```

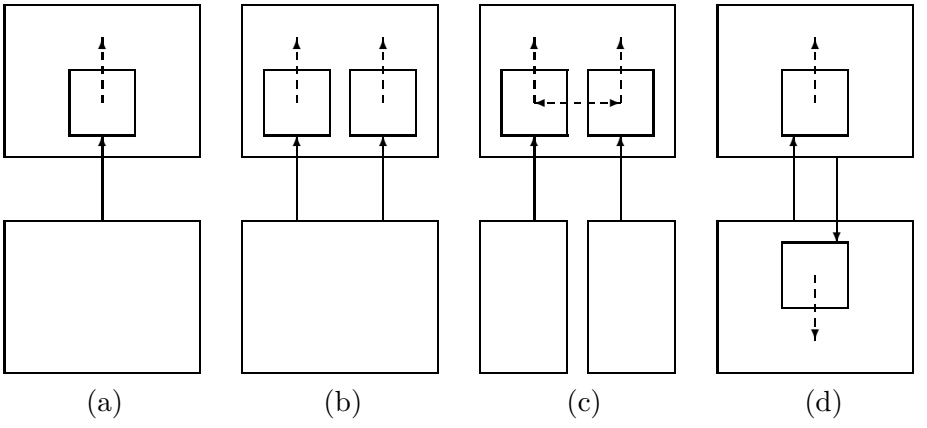


Fig. 1. The possible uses of roles

Notice that in this case two different sessions will be kept: one for `chris` as normal `user` and the other for `chris` as `superuser`. Only when it prints its jobs as a normal `user` the page counter is incremented.

4 Uses of Roles in `powerJava`

In this paper we exploit the language `powerJava` in a new way which allows modelling the agent inspired vision of interaction among objects. The basic idea of `powerJava` is that objects (e.g. `hp8100`), called institutions, are composed of roles which can access the state of the institution and of other sibling roles and, thus, can coordinate with each other [3]. However, since an institution is just an object which happens to contain role implementations, nothing prevents us to consider *every object* as an institution, and to consider the roles as different ways of interacting with it. Many objects can play the same role (a printer can have many users) as well as the same object can play different roles (`chris` is both a user and a superuser). Each role instance has its own state, which represents the state of the interaction with the player of the role.

Figure 1 illustrates the different interaction possibilities given by roles, which do not exclude the traditional direct interaction with the object when roles are not necessary. Other possibilities like sessions shared by multiple objects are not considered for space reasons.

Arrows represent the relations between players and their respective roles, *dashed arrows* represent the access relation between objects, i.e., their powers.

- Drawing (a) illustrates the situation where an object interacts with another one by means of the role offered by it. This is, for instance, the case of `sergio` being a `SuperUser` of `hp8100`.
- Drawing (b) illustrates an object (e.g., `chris`) interacting in two different roles with another one (`hp8100` in the example). This situation is used when

an object implements two different interfaces for interacting with it, which have methods (like `print`) with the same signature but with different meaning. In our model the methods of the interfaces are implemented in the roles offered by the objects to interact with them. The role represent also the different sessions of the interaction with the different objects.

– Drawing (c) illustrates the case of two objects which interact by means of the roles of an institution (which can be considered as the context of execution). This is the original case, `powerJava` has been developed for [3]; in this paper, we used as a running example the well-known 5 philosophers scenario. The institution is the table, at which philosophers are sitting and coordinate to take the chopsticks and eat since they can access the state of each other. The coordinated objects are the players of the role `chopstick` and `philosopher`. The former role is played by objects which produce information, the latter by objects which consume them. None of the players contains the code necessary to coordinate with the others, which is supplied by the roles.
– In drawing (d) two objects interact with each other, each playing a role offered by the other. This is often the case of interaction protocols: e.g., an object can play the role of *initiator* in the Contract Net Protocol if and only if the other object plays the role of *participant*. Indeed, the Contract Net Protocol is reported as an example in the following section.

The four cases can be combined to represent more complex interaction schemas.

This view of roles inspires a new vision of the the OO paradigm, which accounts for the way humans conceptualize objects performed in philosophy and above all in cognitive science [15]. In particular, cognitive science has highlighted that properties of objects are not objective properties of the world, but they depend on the properties of the agent conceptualizing the object: objects are conceptualized on the basis of what they "afford" to the actions of the entities interacting with them. Thus, different entities conceptualize the same object in different ways. We translate this intuition in the fact that an object offers different methods according to which type of object it is calling it: the methods offered (the powers of a role) depend on the requirements offered by the caller.

4.1 The Contract Net Protocol Example

Hereafter, we report an example set in the framework of interaction protocols, describing an implementation of the well-known *contract net* protocol. The example follows the interaction schema (d), reported in the previous section, and it is substantially different than the analogous example reported in a previous paper [4]. In fact, the solution proposed here is *distributed* instead of being *centralized* (let us denote by this name a solution respecting case (c) in the previous section). The advantage of the old solution was that players did not need to know anything about the coordination mechanism. In this case, instead, each object also supplies a role for its counterpart, which describes the powers that

are given to the counterpart in the interaction. For instance, the object that will play the `initiator` role will define the powers of the `participants`, and vice versa. The powers are the messages that the `initiator` will understand; this is very different than our previous proposal, where the powers only allowed to start a negotiation or to take part to a negotiation, depending on the role, and the exchanged messages were hidden inside the institution.

In this new version, roles are also used for maintaining interaction sessions. In the following example, `refuseProposal` can be executed only if `cfp` has already been executed, this can be tracked thanks to the role state and, in particular, thanks to variable `state`.

Observe that when the object, offering a role, is supposed to answer something, it needs to invoke a method, which is supplied as a power of a role, which is in turn offered by the object to which it is responding. In the contract net, a possible answer to a `cfp` is the performative `propose`. In this case, see also the code reported at the end of this section, the above interaction is implemented by the instruction:

```
(that <-(Participant) Peer.this).propose(getProposal(task))
```

Here, `Peer.this` refers to the object offering the role `initiator`; such an object means to play the role of `Participant` and, in particular, to invoke the power `propose` offered by this role. The role `participant` is offered by the object which is currently playing the initiator (identified in the above code line by `that`), see Fig. 2.

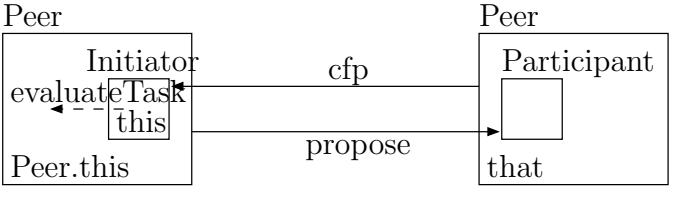


Fig. 2. Description of the interaction between an Initiator and a Participant, when, after a "cfp" performative, the answer will be a "propose" performative

The communication is asynchronous, since the proposal is not returned by the `cfp` method.

Notice that an object which is currently playing the role of participant in a given interaction, can at the same time play the role of initiator in another interaction. See the method `evaluateTask`, in which a new interaction is started for executing a subtask by creating the two roles in the respective objects and by linking players to them:

```
role Initiator playedby InitiatorReq {
  void cfp(Task task);
  void rejectProposal(Proposal proposal);
  void acceptProposal(Proposal proposal);
```

```
}
interface InitiatorReq { // must implement the role specification Participant
}

role Participant playedby ParticipantReq {
  void propose(Proposal proposal);
  void refuse(Task task);
  void inform(Object result);
  void failure(Object error);
}

interface ParticipantReq { // must implement the role specification Initiator
}

class Peer implements ParticipantReq, InitiatorReq
{

  definerole Initiator {
    final static int STATE_1 = 1;
    final static int STATE_2 = 2;
    int state = STATE_1;

    public void cfp(Task task) {
      if (state != STATE_1)
        throws new IllegalPerfomativeException();
      state = STATE_2;
      if (evaluateTask(task))
        (that <-(Participant) Peer.this).propose(getProposal(task));
      else
        (that <-(Participant) Peer.this).refuse(task);
    }

    public void refuseProposal(Proposal proposal) {
      if (state != STATE_2)
        throws new IllegalPerformativeException();
      removeProposal(proposal);
      state = STATE_1;
    }

    public void acceptProposal(Proposal proposal) {
      if (state != STATE_2)
        throws new IllegalPerfomativeException();
      try {
        (that <-(Participant) Peer.this).inform(performTask(proposal, task));
      } catch(TaskExecException err) {
        (that <-(Participant) Peer.this).failure(err);
      }
      state = STATE_1;
    }
```

```
  }

  private boolean evaluateTask(Task task) {
    Task subTask;  // ...
    this.new Participant(peer);
    peer.new Initiator(this);
    (peer <-(Initiator) this).cfp(subTask);   // ...
  }

  definerole Initiator { ... }

}
```

5 Conclusion

In this work, we have proposed the introduction of a form of interaction between objects, in the OO paradigm, which borrows from the theory about agent communication. The main advantage is to allow session-aware interactions in which the history of the occurred method invocations can be taken into account and, thus, introducing the possibility of realizing, in a quite natural way, agent interaction protocols. The key concept which allows communication is the role played by an object in the interaction with another object. Besides proposing a model that describes this form of interaction, we have also proposed an extension of the language `powerJava` that accounts for it.

One might wonder whether the introduction of agent-like communication between objects gives us some feedback to the agent world. We believe that the following lessons can be learnt, in particular, concerning roles:

- Roles must be distinguished in role types and role instances: role instances must be related to the concept of session of an interaction.
- The notion of role is useful not only for structuring institutions and organizations but for dealing with interaction among agents.
- The notion of affordance can be used to allow agents to interacts in different ways with different kind of agents.

In this paper, we show a different way of using `powerJava` exploiting roles to model communications where: the method call specifies the caller of the object, the state of the interaction is maintained, methods can be part of protocols, objects play roles in the interaction and method calls can be asynchronous as in agent protocols.

This proposal builds upon the experience that the authors gathered on the language `powerJava` [3,4,5,6], which is implemented by means of a precompiler. Basically `powerJava` shares the idea of gathering roles inside wider entities with languages like Object Teams [17] and Ceasar [20]. These languages emerge as refinements of aspect oriented languages aiming at resolving practical limitations of other languages. In contrast, our language starts from a conceptual modelling of roles and then it implements the model as language constructs. Differently

than these languages we do not model aspects. The motivation is that we want to stick as much as possible to the Java language. However, aspects can be included in our conceptual model as well, under the idea that actions of an agent playing a role "count as" actions executed by the role itself. In the same way, the execution of methods of an object can give raise by advice weaving to the execution of a method of a role. On the other hand, these languages do not provide the notion of role casting we introduce in `powerJava`. Roles as double face interfaces have some similarities with Traits [22] and Mixins. However, they are distinguished because roles are used to extend instances and not classes. Finally, C# allows for multiple implementations of interfaces. None of the previous works, however, considers the fact that roles work as sessions of the interaction between objects.

By implementing agent like communication in an OO programming language, we gain in simplicity in the language development, importing concepts that have been developed by the agent community inside the Java language itself. This language is, undoubtedly, one of the most successful currently existing programming languages, which is also used to implement agents even though it does not supply specific features for doing it. The language extension that we propose is a step towards the overcoming of these limits.

At the same time, introducing theoretically attractive agent concepts in a widely used language can contribute to the success of the Autonomous Agents and Multiagent Systems research in other fields. Developers not interested in the complexity of agent systems can anyway benefit from the advances in this area by using simple and concrete constructs in a traditional programming language.

Future work concerns making explicit the notion of state of a protocol so to make it transparent to the programmer and allow to define the same method with different meanings in each state. Finally, the integration of centralized and decentralized approaches to coordination among roles (drawings (c) and (d) of Figure 1) must be studied.

References

1. A. Albano, R. Bergamini, G. Ghelli, and R. Orsini. An object data model with roles. In *Procs. of VLDB'93*, pages 39–51, 1993.
2. F. Arbab. Abstract behavior types: A foundation model for components and their composition. In *Formal Methods for Components and Objects, LNCS 2852*, pages 33–70. Springer Verlag, Berlin, 2003.
3. M. Baldoni, G. Boella, and L. van der Torre. Roles as a coordination construct: Introducing powerJava. In *Procs. of MTCoord'05 workshop at COORDINATION'05*, 2005.
4. M. Baldoni, G. Boella, and L. van der Torre. Bridging agent theory and object orientation: Importing social roles in object oriented languages. In *Post-Procs. of PROMAS'05 workshop at AAMAS'05*, volume 3862 of LNCS, pages 57-75, Springer, 2006.

5. M. Baldoni, G. Boella, and L. van der Torre. Powerjava: ontologically founded roles in object oriented programming language. In *Proc. of 21st ACM Symposium on Applied Computing, SAC 2006, Special Track on Object-Oriented Programming Languages and Systems (OOPS 2006)*, pages 1414-1418, Dijon, France, April 2006. ACM.
6. M. Baldoni, G. Boella, and L. van der Torre. Interaction among Objects via Roles – Sessions and Affordances in Java. In *Proc. of the 4th International Conference on Principles and Practices of Programming In Java (PPPJ 2006)*, pages 188-193, Mannheim, Germany, 2006).
7. B. Bauer, J.P. Muller, and J. Odell. Agent UML: A formalism for specifying multiagent software systems. *Int. Journal of Software Engineering and Knowledge Engineering*, 11(3):207–230, 2001.
8. F. Bellifemine, A. Poggi, and G. Rimassa. Developing multi-agent systems with a FIPA-compliant agent framework. *Software - Practice And Experience*, 31(2):103–128, 2001.
9. G. Boella and L. van der Torre. Attributing mental attitudes to roles: The agent metaphor applied to organizational design. In *Procs. of ICEC'04*. IEEE Press, 2004.
10. G. Boella and L. van der Torre. A game theoretic approach to contracts in multiagent systems. *IEEE Transactions on Systems, Man and Cybernetics - Part C*, 2006.
11. G. Boella and L. van der Torre. Security policies for sharing knowledge in virtual communities. *IEEE Transactions on Systems, Man and Cybernetics - Part A*, 2006.
12. M. Dahchour, A. Pirotte, and E. Zimanyi. A generic role model for dynamic objects. In *Procs. of CAiSE'02*, volume 2348 of *LNCS*, pages 643–658. Springer, 2002.
13. M. Dastani, V. Dignum, and F. Dignum. Role-assignment in open agent societies. In *Procs. of AAMAS'03*, pages 489–496, New York (NJ), 2003. ACM Press.
14. J. Ferber, O. Gutknecht, and F. Michel. From agents to organizations: an organizational view of multiagent systems. In *LNCS n. 2935: Procs. of AOSE'03*, pages 214–230. Springer Verlag, 2003.
15. J. Gibson. *The Ecological Approach to Visual Perception*. Lawrence Erlabum Associates, New Jersey, 1979.
16. N. Guarino and C. Welty. Evaluating ontological decisions with ontoclean. *Communications of ACM*, 45(2):61–65, 2002.
17. S. Herrmann. Object teams: Improving modularity for crosscutting collaborations. In *Procs. of Net.ObjectDays*, 2002.
18. T. Juan and L. Sterling. Achieving dynamic interfaces with agents concepts. In *Procs. of AAMAS'04*, 2004.
19. C. Masolo, L. Vieu, E. Bottazzi, C. Catenacci, R. Ferrario, A. Gangemi, and N. Guarino. Social roles and their descriptions. In *Procs. of KR'04*, pages 267–277. AAAI Press, 2004.
20. M. Mezini and K. Ostermann. Conquering aspects with caesar. In *Procs. of the 2nd International Conference on Aspect-Oriented Software Development (AOSD)*, pages 90–100. ACM Press, 2004.
21. M.P. Papazoglou and B.J. Kramer. A database model for object dynamics. *The VLDB Journal*, 6(2):73–96, 1997.
22. N. Scharli, S. Ducasse, O. Nierstrasz, and A. Black. Traits: Composable units of behavior. In Springer Verlag, editor, *LNCS, vol. 2743: Procs. of ECOOP'03*, pages 248–274, Berlin, 2003.

23. F. Steimann. On the representation of roles in object-oriented and conceptual modelling. *Data and Knowledge Engineering*, 35:83–848, 2000.
24. M. Winikoff. JACK - intelligent agents: An industrial strength platform. In R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors, *Multi-Agent Programming*, pages 175–193. Springer Verlag, Berlin, 2005.
25. M. J. Wooldridge and N. R. Jennings. Intelligent agents: Theory and practice. *Knowledge Engineering Review*, 10(2):115–152, 1995.
26. F. Zambonelli, N.R. Jennings, and M. Wooldridge. Developing multiagent systems: The Gaia methodology. *IEEE Transactions of Software Engineering and Methodology*, 12(3):317–370, 2003.

Adding Knowledge Updates to 3APL

Vivek Nigam* and João Leite

CENTRIA, New University of Lisbon,Portugal
vivek.nigam@gmail.com, jleite@di.fct.unl.pt

Abstract. 3APL is a widely known multi-agent programming language. However, when to be used in certain domains and environments, 3APL has some limitations related to its simplistic update operator that only allows for updates to the extensional part of the belief base and its lack of a language with both default and strong negation to enable the representation and reasoning about knowledge with the open and closed world assumptions. In this paper, we propose to address these issues by modifying the belief base of 3APL to be represented by Dynamic Logic Programming, an extension of Answer-Set Programming that allows for the representation of knowledge that changes with time.

1 Introduction

In the past few years, several agent architectures and agent programming languages have been proposed. Among them we can find, for example, 3APL [9,12], FLUX [22], IMPACT [10], DALI [8], JASON [5] and Minerva [14,18]. For a survey on some of these systems, as well as others, see [6,7,20].

In this paper, we take a closer look at 3APL, one of the existing systems that has recently received an increasing amount of attention, and propose some enhancements to its language and semantics.

3APL is a logic based programming language for implementing cognitive agents that follows the classical BDI architecture where agents have beliefs (B), intentions (I) and desires (D) to guide their actions. The semantics of 3APL agents is defined by a *transition system* composed of *transition rules*. The use of 3APL provides the agent programmer with a very *intuitive* and simple way to define agents. The programmer can declaratively specify the *beliefs* (represented by Horn Clauses) and *goals* (represented by conjunctions of atoms) of agents, how they *build plans* to achieve such goals, and reason with their beliefs. Furthermore, communication between agents can be done in an elegant way by modifying the beliefs of agents, allowing for the possibility of reasoning with the transferred messages. Despite all these interesting properties, 3APL, when to be used in certain domains and environments, has some limitations that serve as our motivation to propose the modifications presented in this paper. These limitations, in our opinion, are:

* Supported by the Alβan Program, the European Union Programme of High Level Scholarships for Latin America, no. E04M040321BR.

R.H. Bordini et al. (Eds.): ProMAS 2006, LNAI 4411, pp. 165–181, 2007.

1. Limited belief updates - The mechanism used by 3APL to update agent's beliefs is quite limited. Such updates in 3APL amount to the simple addition and removal of facts in the agent's belief base. It is not difficult to find a situation where this type of belief update is insufficient. Consider an agent with a belief base containing the rule $believe(santa\_claus) \leftarrow mother\_said(santa\_claus)$, and the fact $mother\_said(santa\_claus)$. This agent can be seen as a child agent that believes in everything its mother says, in this case it believes in *santa claus*, because its mother said so ($mother\_said(santa\_claus)$). Furthermore, consider that the agent evolves and discovers that in fact, *santa claus* doesn't exist, even though its mother said so. Since 3APL only allows for updates to the extensional part of the belief base (i.e. its set of facts), it is not possible to achieve the desired semantics, where $believe(santa\_claus)$ is false and $mother\_said(santa\_claus)$ is true, by the mere addition and retraction of facts. Note that it is not possible to remove the fact $believe(santa\_claus)$ because there is none to be removed, and if the fact $mother\_said(santa\_claus)$ is removed it would be change the belief base in an undesired way. To obtain the desired effect, updates in the intensional part of the knowledge base (i.e. its set of rules) are required;

2. Limited expressive power of negative information - 3APL allows for the use of one form of negation, namely *negation by finite failure*. It has been shown that the use of default negation (*not*) provides good expressive power to a language. Furthermore, the use of both default and strong negations ($\neg$), concurrently, such as in Answer-Set Programming [11], allows for easy ways to reason with both the *closed* and *open world assumptions*. For example, in the classical car-train cross, where the car should pass the cross if its sure that the train is not coming, it is necessary to reason with the open world assumption, where strong negation plays a key role ($\neg train$). On the other hand, to represent a cautious agent that would move if it believes that a place is not safe ($not\ safe$), the use of default negation is more adequate;

In this paper, we will use Dynamic Logic Programming (DLP) [19,2,14], an extension of Answer Set Programming, to address these limitations stated above. We propose to represent the 3APL agent's *belief base* as a Dynamic Logic Program.

According to the paradigm of *DLP*, knowledge is given by a series of theories, encoded as generalized logic programs[1], each representing distinct states of the world. Different states, sequentially ordered, can represent different time periods, thus allowing DLP to represent knowledge that undergoes successive updates. Since individual theories may comprise mutually contradictory as well as overlapping information, the role of *DLP* is to employ the mutual relationships among different states to determine the declarative semantics for the combined theory comprised of all individual theories at each state. Intuitively, one can add, at the end of the sequence, newer rules (arising from new or reacquired knowledge) leaving to *DLP* the task of ensuring that these rules are in force, and that previous ones are valid (by inertia) only so far as possible, i.e. that

[1] Logic programs with default and strong negation both in the body and head of rules.

they are kept for as long as they are not in conflict with newly added ones, these always prevailing.

By using DLP to represent the agent's *belief base*, we address, at once, both of the limitations stated above. The first, namely the one related to the scope of the existing 3APL update operator, is immediately solved by the very foundational scope of DLP, after 3APL is adapted to accommodate such change. With DLP, 3APL agents will be able to maintain an *up to date belief base* in situations where both the extensional and intensional parts of the knowledge base change. Agent's simply have to add, at the end of the sequence of programs that constitutes their *belief base*, new facts and rules alike, and not worry with emerging contradictions with previous rules as the DLP semantics properly handles them. The second limitation is also addressed by using DLP, as the object language used to define the generalized logic programs allows for both default and strong negations, inherited from Answer-Set Programming [11] that it generalizes.

En passant, we take the opportunity provided by the fact that DLP allows for rule based updates, to also increase the expressiveness of the messages transmitted between the agents, by allowing their content to consist of generalized logic programs. By transmitting logic programs, instead of facts, agents will be able to exchange knowledge containing rules. Depending on its beliefs, the receiving agent can update its beliefs by the transmitted logic program, thus facilitating learning (through rule teaching).

This remainder of the paper is distributed in the following way. We begin in the Section 2 to give some preliminary definitions related to 3APL and Dynamic Logic Programming that will be used throughout the paper. Later, in Section 3, we modify the syntax of some of the transition rules of 3APL. In Section 4 we present the semantics of the belief query language and of the proposed transition rules. In Section 5, we discuss some of the added features obtained by the modification proposed and in Section 6 we give an illustrative example with some of the properties of the modified system. Finally, in Section 7 we conclude with some suggestions of further investigation.

2 Preliminaries

In this Section, after introducing some concepts of *logic programs*, we introduce the semantics of *Dynamic Logic Programs* and *partially* introduce the *3APL multi agent language* in its propositional form. For the sake of space, we are only going to introduce the reader the definitions of 3APL that are *relevant* for this paper, further details about the complete version of 3APL system can be found in [9].

2.1 Languages and Logic Programs

Let $\mathcal{K}$ be a set of propositional atoms. An *objective literal* is either an atom A or a strongly negated atom $\neg A$. A *default literal* is an objective literal preceded by *not*. A *literal* is either an objective literal or a default literal. We also define

the set of objective literals $\mathcal{L}_{\mathcal{K}}^{\neg} = \mathcal{K} \cup \{\neg A \mid A \in \mathcal{K}\}$ and the set of literals $\mathcal{L}_{\mathcal{K}}^{\neg,not} = \mathcal{L}_{\mathcal{K}}^{\neg} \cup \{not\ L \mid L \in \mathcal{L}_{\mathcal{K}}^{\neg}\}$ over the alphabet $\mathcal{K}$. We are going to use the set *Disjunction* to build the belief query language, $\mathcal{L}_B$, if $A \in \mathcal{K}$ then $B(A), \neg B(A) \in$ *Disjunction*, $\top \in$ *Disjunction* and if $\delta, \delta' \in$ *Disjunction* then $\delta \vee \delta' \in$ *Disjunction*, if $\delta \in$ *Disjunction* then $\delta \in \mathcal{L}_B$, furthermore if $\phi, \phi' \in \mathcal{L}_B$ then $\phi \wedge \phi' \in \mathcal{L}_B$. Informally, $\mathcal{L}_B$ is the smallest set containing all the formulas in conjunction normal form, where $B(.)$ and $\neg B(.)$ are the *literals* of the language. The goal query language, $\mathcal{L}_G$, is defined the following way, $\top \in \mathcal{L}_G$, if $A \in \mathcal{K}$ then $G(\phi) \in \mathcal{L}_G$, and if $k, k' \in \mathcal{L}_G$ then $k \wedge k' \in \mathcal{L}_G$.

A *rule* r is an ordered pair $Head\,(r) \leftarrow Body\,(r)$ where $Head\,(r)$ (dubbed the head of the rule) is a literal and $Body\,(r)$ (dubbed the body of the rule) is a finite set of literals. A rule with $Head\,(r) = L_0$ and $Body\,(r) = \{L_1, \ldots, L_n\}$ will simply be written as $L_0 \leftarrow L_1, \ldots, L_n$. A *generalized logic program* (*GLP*) P, in $\mathcal{K}$, is a finite or infinite set of rules. If $Head(r) = A$ (resp. $Head(r) = not\ A$) then $not\ Head(r) = not\ A$ (resp. $not\ Head(r) = A$). If $Head\,(r) = \neg A$, then $\neg Head\,(r) = A$. By the *expanded generalized logic program* corresponding to the GLP P, denoted by $\mathbf{P}$, we mean the GLP obtained by augmenting P with a rule of the form $not\ \neg Head\,(r) \leftarrow Body\,(r)$ for every rule, in P, of the form $Head\,(r) \leftarrow Body\,(r)$, where $Head\,(r)$ is an objective literal[2]. Two rules r and r' are conflicting, denoted by $r \bowtie r'$, iff $Head(r) = not\ Head(r')$. An *interpretation* M of $\mathcal{K}$ is a set of objective literals that is consistent i.e., M does not contain both A and $\neg A$. An objective literal L is true in M, denoted by $M \vDash L$, iff $L \in M$, and false otherwise. A default literal $not\ L$ is true in M, denoted by $M \vDash not\ L$, iff $L \notin M$, and false otherwise. A set of literals B is true in M, denoted by $M \vDash B$, iff each literal in B is true in M. An interpretation M of $\mathcal{K}$ is an *answer set* of a GLP P iff $M' = least\,(\mathbf{P} \cup \{not\ A \mid A \notin M\})$, where $M' = M \cup \{not\_A \mid A \notin M\}$, A is an objective literal, and $least(.)$ denotes the least model of the definite program obtained from the argument program by replacing every default literal $not\ A$ by a new atom not_A. For notational convenience, we will no longer explicitly state the alphabet $\mathcal{K}$. We will consider the alphabet of the language at an instant, consisting precisely of all the propositional symbols that appear *explicitly* in the program at such instant. Therefore, the alphabet of a program may change if new propositional symbols are included in the program. Furthermore, as usual, we will consider all the variables appearing in the programs as a shorthand for the set of all its possible ground instantiations.

2.2 Dynamic Logic Programming

A *dynamic logic program* (*DLP*) is a sequence of generalized logic programs. Let $\mathcal{P} = (P_1, ..., P_s)$, $\mathcal{P}' = (P'_1, ..., P'_n)$ and $\mathcal{P}'' = (P''_1, ..., P''_s)$ be DLPs. We use $\rho\,(\mathcal{P})$ to denote the multiset of all rules appearing in the programs $\mathbf{P}_1, ..., \mathbf{P}_s$, and $(\mathcal{P}, \mathcal{P}')$ to denote $(P_1, ..., P_s, P'_1, ..., P'_n)$, and $(\mathcal{P}, P'_1)$ to denote $(P_1, ..., P_s, P'_1)$.

[2] Expanded programs are defined to appropriately deal with strong negation in updates. For more on this issue, the reader is invited to read [15,14]. In subsequent sections, and unless otherwise stated, we will always consider generalized logic programs to be in their expanded versions.

Each position, i, of sequence of programs that constitutes a DLP, represents a state of the world (for example different time periods), and the corresponding logic program in the sequence, P_i, contains some knowledge that is supposed to be true at this state. The role of Dynamic Logic Programming is to assign a semantics to the combination of these possibly contradictory programs, by using the mutual relationships existing between them. This is achieved by considering only the rules that are not conflicting with rules in a GLP that is in a position ahead in the sequence of programs. Intuitively, one could add a new GLP to the end of the sequence, representing a new update to the knowledge base, and let DLP solve, automatically, the possible contradictions originated by this new update.

Definition 1 (Semantics of DLP). *[14,1] Let $\mathcal{P} = (P_1, \ldots, P_s)$ be a dynamic logic program over language $\mathcal{K}$, A an objective literal, $\rho(\mathcal{P})$, M' and least(.) as before. An interpretation M is a stable model of $\mathcal{P}$ iff*

$$M' = least\left([\rho(\mathcal{P}) - Rej(M, \mathcal{P})] \cup Def(M, \mathcal{P})\right)$$

Where:

$$Def(M, \mathcal{P}) = \{not\, A \mid \nexists r \in \rho(\mathcal{P}), Head(r) = A, M \vDash Body(r)\}$$
$$Rej(M, \mathcal{P}) = \{r \mid r \in \mathbf{P}_i, \exists r' \in \mathbf{P}_j, i \leq j \leq s, r \bowtie r', M \vDash Body(r')\}$$

We can use DLPs to elegantly represent evolving knowledge base, since their semantics is defined by using the whole history of updates and by giving a higher priority to the newer information. We will illustrate how this is achieved in the following example.

Example 1. Consider a DLP, $\mathcal{P}$, that initially contains only the program P_1, with the intended meaning that: if the tv is on (tv_on) the agent will be watching the tv ($watch\_tv$); if the tv is off it will be sleeping ($sleep$); and that the tv is currently on.

$$\begin{aligned} P_1 : \; & sleep \leftarrow not\, tv\_on \\ & watch\_tv \leftarrow tv\_on \\ & tv\_on \leftarrow \end{aligned}$$

The DLP has as expected, only one stable model, namely $\{watch\_tv, tv\_on\}$, where the agent is watching tv and not sleeping.

Consider now that $\mathcal{P}$ is updated by the program P_2, stating that if there is a power failure ($power\_failure$) the tv cannot be on, and that currently there is a power failure.

$$\begin{aligned} P_2 : \; & not\, tv\_on \leftarrow power\_failure \\ & power\_failure \leftarrow \end{aligned}$$

Since the program P_2 is newer than the previous program P_1, the rule, $tv\_on \leftarrow$, will be rejected by the rule $not\, tv\_on \leftarrow power\_failure$. Thus obtaining the expected stable model $\{sleep, power\_failure\}$, where the agent is sleeping and

the tv is no longer on. Furthermore, consider one more update stating that the power failure ended.

$$P_3 : not\, power\_failure \leftarrow$$

Because of the update P_3, the rule $\{power\_failure \leftarrow\} \subset P_2$ is rejected and $power\_failure$ should not be considered as true. Therefore, the rule $\{tv\_on \leftarrow\} \subset P_1$ is no longer rejected, and again the agent will conclude that the tv is on and it did not fall asleep. As expected, the stable model of the updated program is once more $\{watch\_tv, tv\_on\}$.

Of course, due to the lack of interesting programs on the television, it might happen that we don't watch tv even if the tv is on. We can use a new update, P_4, to represent this situation:

$$\begin{aligned} P_4 : \; & not\, watch\_tv \leftarrow bad\_program \\ & good\_program \leftarrow not\, bad\_program \\ & bad\_program \leftarrow not\, good\_program \end{aligned}$$

With this new update the DLP will have two stable models, one considering that the tv show is good and the agent is watching the tv ($\{good\_program, watch\_tv, tv\_on\}$), and another that the program is bad and it is not watching the tv ($\{bad\_program, tv\_on\}$).

As illustrated in the example above, a DLP can have more than one stable model. But then how to deal with these stable models and how to represent the semantics of a DLP? This issue has been extensively discussed and three main approached are currently being considered [14]:

Skeptical - $\models_\cap$ According to this approach, the *intersection* of all stable models is used to determine the semantics of a DLP. As we are going to represent the beliefs of the agent as a DLP, this approach would be best suited for more skeptical agents, since they would only believe in a statement if all stable models (possible worlds) support this statement;

Credulous - $\models_\cup$ According to this approach, the *union* of all stable models is used to determine the semantics of a DLP. With this semantics, a DLP would consider as true all the objective literals that are true in one of its stable models;

Casuistic - $\models_\Omega$ According to this approach, one of the stable models is selected, possibly by a selection function Ω, to represent the semantics of the program. Since the stable models of a belief base can be seen as representations of possible worlds, an agent using this approach would commit to one of them to guide their actions.

We will denote by $SM(\mathcal{P})$ the set of all stable models of the DLP $\mathcal{P}$. Further details and motivations concerning DLPs and its semantics can be found in [14].

2.3 Propositional 3APL

The 3APL agent is composed of a *belief base* (σ) that represents how the world is for the agent, a *goal base* (γ) representing the set of states that the agent

wants the world to be, a set of *capabilities* (Cap) that represents the set of actions the agent can perform, a *plan base* (Π) representing the plans that the agent is performing to achieve specific goals, sets of *goal planning rules and plan revision rules* (PG, PR) that are used by the 3APL agent to build plans, and the *environment* (ξ) in which the agent is situated. The environment can be viewed as a set of facts.

The belief base of the agent is composed of a set of rules of the form, $(A \leftarrow A_1, \ldots, A_n)$, where $A_1, \ldots, A_n, A \in \mathcal{K}$. The goal base is composed by a set containing sets of atoms[3], $\{\Sigma_1, \ldots \Sigma_n \mid \Sigma_i \subseteq \mathcal{K}, 1 \leq i \leq n\}$. Each set contained in the goal base will represent a goal of the agent. For example in the classical block world problem, if the goal base of an agent contains the set of atoms $\{on(A, B), on(C, D)\}$, a goal of the agent would be to have the block A over the block B ($on(A, B)$), and simultaneously the block C over the block D ($on(C, D)$).

Plans can be composed of several types of actions (*communication action*, *mental action*, *external action*, *test action*, *composite plans*, etc). We are interested in the *mental actions* and *communication actions*. We will denote the empty plan as ϵ. We will not formally define the language of plans ($\mathcal{L}_{\mathcal{P}}$), since it will not be extensively used in this paper, more interested readers are invited to read [9].

Definition 2 (Mental Actions Specifications). *[9] Let $\beta \in \mathcal{L}_B$ be the precondition of the mental action, α be a mental action name, $Lit_B = \{B(\phi), \neg B(\phi) \mid \phi \in \mathcal{K}\}$ and $\beta' = \beta_1 \wedge, \ldots, \wedge \beta_n$ be the postcondition of the mental action, where $\beta_1, \ldots, \beta_n \in Lit_B$ Then, a mental action is a tuple $\langle \beta, \alpha, \beta' \rangle$, and Mact is the set of all mental actions.*

The communication actions are represented by the special predicate $Send(r, type, A)$, where r is the name of the agent the message is being sent to, *type* is the performative indicating the nature of the message and the message $A \in \mathcal{K}$.

The semantics of an agent in 3APL is given by *transition rules*. We will be concerned in this paper with two transition rules corresponding to the *mental* and the *communication actions*.

Since the set of capabilities and revision rules that an agent maintains is the same throughout time, we can define the concept of *agent configuration* which is used to represent (the variable part of) the *state of an agent* at a given time. We simplify the definition given in [9] to the propositional version of 3APL.

Definition 3 (Agent Configuration). *[9] An agent configuration is represented by the tuple $\langle \sigma, \gamma, \Pi \rangle$, where σ is the agent's Belief Base. γ is the agent's Goal Base, such that for any ϕ such that $\gamma \models \phi$, we have that $\sigma \not\models \phi$. $\Pi \subseteq \mathcal{L}_P \times \mathcal{L}_G$ is the plan base of the agent.*

The semantics of the belief and goal query formulas entailment in the propositional 3APL is quite straightforward and will not be *explicitly* defined. The reader is invited to read [9] for further information.

[3] We differ from the notation used in [9], where the conjunction symbol $\wedge$ is used to represent the conjunction of goals.

As mentioned earlier, the agent uses the mental actions to update its beliefs. The update of the belief base of the 3APL agent is done in a quite simple way, by *removing* or *adding* facts to the belief base. Informally, when the precondition β, of a mental action $\langle \beta, \alpha, \beta' \rangle$, is believed by the agent, it will add, as a fact in its belief base, the literals in the postcondition β' that are not negated and remove the ones that are negated. The formal definition can be found in [9].

After performing a communication action $Send(r, type, A)$, a fact, *sent(r,type, A)*, stating that a message A, of type *type*, was sent to the agent r, is included in the belief base of the sending agent. A similar fact, *received(s,type,A)*, is included in the receiving agent's belief base, stating that a message A of type *type* was send by the agent *s*. Notice that messages exchanged between agents are only positive atoms, as no rules can be communicated.

An agent in 3APL uses its *Reasoning Rules* to adopt or change plans. There are two types of Reasoning Rules: the *Goal Planning Rules* and the *Plan Revision Rules*, the former being used by the agent to pursue a new goal and build a initial plan, and the later being used to revise a previously existing plan to obtain another plan. It maybe possible that one or more rules are applicable in a certain agent configuration, and the agent must decide which one to apply. In 3APL this decision is made through a *deliberation cycle*. Further details about the deliberation cycle can also be found in [9]. Here, we will not deal with the 3APL reasoning rules.

3 Modified Syntax

In this Section, we are going to begin to address the 3APL's limitations that we discussed previously, namely its limited capacity of updating an agent's beliefs and its limited expressive power of negative information. We introduce in the following definitions the syntax of the modified 3APL that we propose.

We begin modifying the agent configuration, by replacing the old belief base (σ) by a DLP. The goal base (γ) and the plan base (Π) are as in the original agent configuration.

Definition 4 (Modified Agent Configuration). *The Modified Agent Configuration is the tuple $\langle \sigma, \gamma, \Pi \rangle$, where σ is a DLP representing the agent's belief base. γ, Π are as before, representing, respectively, the agent's goal base and plan base.*

Now we modify the 3APL belief query language, by incorporating two types of negation, *negation by default* and *strong negation*. This will make it possible for the agent to reason with the open and closed world assumptions as we will investigate in the next Section.

Definition 5 (Modified Belief Query Language). *Let $\phi \in \mathcal{L}^{\neg, not}$. The modified belief query language, $\mathcal{L}_{\mathcal{B}}^{\mathcal{M}}$ is defined as follows:*

- $\top \in \mathcal{L}_{\mathcal{B}}^{\mathcal{M}}$;
- $B(\phi) \in \mathcal{L}_{\mathcal{B}}^{\mathcal{M}}$;
- $\beta_M, \beta'_M \in \mathcal{L}_{\mathcal{B}}^{\mathcal{M}}$ *then* $\beta_M \wedge \beta'_M \in \mathcal{L}_{\mathcal{B}}^{\mathcal{M}}$;
- $\beta_M, \beta'_M \in \mathcal{L}_{\mathcal{B}}^{\mathcal{M}}$ *then* $\beta_M \vee \beta'_M \in \mathcal{L}_{\mathcal{B}}^{\mathcal{M}}$.

Notice that differently from the Belief Query formulas in 3APL, the modified queries don't include symbols like $\neg B(\phi)$. As we will discuss with more details in the next Section, we don't feel the need for these type of symbols since the belief operator, $B(.)$, can have a literal, ϕ, as a parameter and not only an atom.

Now that we are considering the belief base of the agent as a Dynamic Logic Program, we will be able to update the belief base with a Generalized Logic Program. As in 3APL, the agent uses mental actions to update its belief base, but we will now consider the postcondition of these actions to be a Generalized Logic Program.

Definition 6 (Modified Mental Actions Specifications). *Let α_M be a modified mental action name, $\beta_M \in \mathcal{L}_{\mathcal{B}}^{\mathcal{M}}$ be the precondition of the action and P a GLP. Then, a modified mental action is a tuple $\langle \beta, \alpha_M, P \rangle$, and $ModAct$ is the set of all modified mental actions.*

$\langle B(tv\_on), turn\_off, \{not\, tv\_on \leftarrow\} \rangle$ is an example of a modified mental action representing the action of turning off the tv. Throughout this paper, we will explore the possibilities of using theses type of actions and give many other examples of application.

In a similar way, we modify the syntax of the communication actions by considering that the message in these actions are GLPs.

Definition 7 (Modified Communication Actions Specifications). *Let s be an agent name, type a performative or speech act and P a GLP. Then, a modified communication action is defined as $Send\,(s, type, P)$, and $ComAct$ as the set of all modified communication actions.*

$Send\,(user, inform, \{not\, power\_failure \leftarrow\})$ is an example of the modified communication action informing the *user* agent that the *power failure* ended.

4 Modified Semantics

In this Section, we define the semantics of the modified system, beginning with the semantics of the *belief query* formulas and afterwards of the *modified mental* and *communication* actions.

4.1 Modified Belief Query Semantics

The semantics of the Belief Queries will depend on the type of approach the agents adopt to handle the multiple stable models of a DLP. As we discussed previously, we consider three approaches: *Skeptical* ($\models_\cap$), *Credulous* ($\models_\cup$) and *Casuistic* ($\models_\Omega$). The consequences of choosing anyone of theses approaches are

not completely clear. More investigation will be needed to determine exactly in what conditions would be more suitable to select one of them, and therefore, we leave the belief query semantics conditioned to the approach used to determine the valuation of the agent's belief base.

Definition 8 (Semantics of Modified Belief Queries). *Let* $B(\phi), \beta_M, \beta'_M \in \mathcal{L}_{\mathcal{B}}^{\mathcal{M}}$ *be belief query formulas,* $\langle \sigma, \gamma, \Pi \rangle$ *be a modified agent configuration and* $x \in \{\cap, \cup, \Omega\}$ *. Then, the semantics of belief query formulas,* $\models_B$*, is defined as follows:*

$$\langle \sigma, \gamma, \Pi \rangle \models_B \top$$
$$\langle \sigma, \gamma, \Pi \rangle \models_B B(\phi) \Leftrightarrow \sigma \models_x \phi$$
$$\langle \sigma, \gamma, \Pi \rangle \models_B \beta_M \wedge \beta'_M \Leftrightarrow \langle \sigma, \gamma, \Pi, \Omega \rangle \models_B \beta_M \text{ and } \langle \sigma, \gamma, \Pi, \Omega \rangle \models_B \beta'_M$$
$$\langle \sigma, \gamma, \Pi \rangle \models_B \beta_M \vee \beta'_M \Leftrightarrow \langle \sigma, \gamma, \Pi, \Omega \rangle \models_B \beta_M \text{ or } \langle \sigma, \gamma, \Pi, \Omega \rangle \models_B \beta'_M$$

We don't feel the need to include in the belief query language the negation of belief literals,$\neg B\,(\phi)$, since with the definition above, the programmer has the possibility of using the open the closed world assumptions by using query formulas of the type $B\,(\neg\phi)$ and $B\,(not\,\phi)$, respectively. Consider the following illustrative example:

Example 2. Let the belief base of an agent consist of the following facts:

$$\{p(a) \leftarrow \qquad\qquad p(b) \leftarrow\}$$

If in the original 3APL, we propose the query $\neg B\,(p\,(c))$ it will succeed, since it is not possible to *unify* $p(c)$ with any of the given facts, and the negation by finite failure will succeed. Hence, the 3APL agents use the closed world assumption.

This query could be done in a similar way in the modified 3APL, by using the modified belief query $B\,(not\,p\,(c))$. As the program above has an unique stable model, namely $\{p(a), p(b)\}$, it would represent the beliefs of the agent. Reminding the definition of the entailment of the default negation: if $\phi \notin M$ then $M \models not\,\phi$. The modified belief query will also succeed, since $p(c) \notin \{p(a), p(b)\}$.

4.2 Semantics of Action Execution

In this subsection we formalize the semantics of the actions in this modification of 3APL.

We start with a definition that formalizes the semantics of the Modified Mental Actions. Informally, if the precondition (β_M) of the modified mental action (α_M) is *satisfied* by the agent configuration, the belief base of the agent will be *updated* with the program (P) in the postcondition of the action. Syntactically, this update adds a new program at the end of the sequence of programs, that composes the Belief Base. We then use the semantics of Dynamic Logic Programming to characterize this updates.

Definition 9 (Semantics of Modified Mental Actions). *Let* $\langle \beta_M, \alpha_M, P \rangle$*,* $\langle \sigma, \gamma, \Pi \rangle$*, be, respectively, a modified mental action and modified agent*

configuration, $x \in \{\cap, \cup, \Omega\}$*, and* $\kappa \in \mathcal{L}_G$ *The semantics of the modified mental action is given by the transition rule:*

$$\frac{\langle \sigma, \gamma, \{(\alpha_M, \kappa)\}\rangle \models_B \beta_M}{\langle \sigma, \gamma, \{(\alpha_M, \kappa)\}\rangle \rightarrow \langle (\sigma, P), \gamma', \{(\epsilon, \kappa)\}\rangle}$$

where $\gamma' = \gamma \setminus \{\Sigma \mid \Sigma \subseteq \mathcal{K} \wedge (\sigma, P) \models_x \Sigma\}$.

This modification in the definition of Mental Action greatly increases the *expressiveness* of the language. Now the agent can use generalized logic programs instead of simple facts to update the belief base. Furthermore, the semantics of DLPs gives us an *intuitive solution* for the conflicting cases, by automatically rejecting older rules if they are *conflicting* with a newer ones.

For example, consider again the situation explained in the Introduction, where the agent has a belief base consisting of the program:

$$mother\_said(santa\_claus) \leftarrow$$
$$believe(santa\_claus) \leftarrow mother\_said(santa\_claus)$$

And after a mental action it would have to conclude that $believe(santa\_claus)$ is no longer true. This can be easily done by updating the belief base with the program $\{not\ believe(santa\_claus) \leftarrow\}$. Then, the DLP semantics will reject the rule $believe(santa\_claus) \leftarrow mother\_said(santa\_claus)$ and the agent will no longer believe in $believe(santa\_claus)$ but still believe in $mother\_said(santa\_claus)$.

Even though we believe that the semantics of DLP can handle most of the conflicting cases in an elegant manner, there are some cases that require *program revision*. Note that revision and updates are two different forms of belief change [13]. To achieve both forms of belief change, would be necessary to include a mechanism that would make it possible for the programmer to customize the revision of the programs, for example, by programming the deliberation cycle. We will not approach this issue in this paper.

The final modification that we propose for the 3APL actions concerns the communication actions. In 3APL the agent uses communication actions to send messages to other agents in the system. Up to now the messages that the agents transmit are positive facts. Since our agents have the possibility to update their beliefs with GLPs, it makes sense to use this added expressiveness and allow GLPs to be exchanged between the agents. Accordingly, in this proposal, the agents will exchange messages containing Generalized Logic Programs.

In a similar way as done in 3APL, after performing a communication action $(Send(r, type, P))$, the sending agent (s) will update its belief base with the program $\{sent(r, type, P) \leftarrow\}$ and the receiving agent (r) updates its belief base with the program $\{received(r, type, P) \leftarrow\}$[4].

By combining modified communication and mental actions, agents are now able to update their belief base with knowledge that they receive. Normally, an

[4] Programs can be associated with identifiers to be used when the facts sent(.) or received(.) are added in the belief base to represent these programs.

agent has a *social point of view* about the other agents in the environment, and may consider the information passed by another *trustworthy* agent to be true. For example, it is usually the case that a son believes what his father tells him. This could be represented using the following modified mental action:

$$\langle \{B(received(father, command, P)) \wedge B(obey(father))\}, obey\_father, \{P\} \rangle$$

where the agent would update its belief base with the program P, if it believes that it should obey his father and that it received from his father a command containing the message P.

5 Properties of the Modified 3APL

In this Section, we elaborate on the features provided by the modification of the 3APL system proposed in this paper.

Evolving Knowledge Bases - By adopting their belief bases as Dynamic Logic Programs and using its semantics to solve the possible conflicts when updating its beliefs, 3APL agents can have *evolving belief bases*. This dynamic character of its knowledge base opens the possibility of performing more complex updates using generalized logic programs instead of adding or removing facts. Agents with this modification can learn new rules even though they *partially* conflict with previous knowledge. For example, an agent may consider that all the published papers are good, represented by the GLP $\{papers\_good(X) \leftarrow\}$. Then, it learns that not all papers are good because the ones published in poor venues are not so good, hence updates its beliefs with the program $\{not\ papers\_good(X) \leftarrow bad\_congress(X)\}$. Notice that if the agent *doesn't believe* the paper X is from a *bad congress* it will use the previous knowledge and consider the *paper as good*. However if it *believes* that the paper X comes from a *bad congress* the newer rule will reject the older one. More about evolving knowledge bases can be found in [14];

The next proposition states that in fact, all DLPs can be semantically represented by an agent in the modified 3APL.

Proposition 1. *Let $\mathcal{P}$ be a DLP, $x \in \{\cap, \cup, \Omega\}$ and $\langle \mathcal{P}, \gamma, \Pi \rangle$ be a modified agent configuration. Then:*

$$(\forall L \in \mathcal{L}^{\neg, not}).(\mathcal{P} \models_x L \Leftrightarrow \langle \mathcal{P}, \gamma, \Pi \rangle \models_x B(L))$$

Proof: Trivial from the definition of the modified belief queries.

Strong and Default Negation - Agents in 3APL treat negation as *negation by failure*. In the modification proposed in this paper, we increase considerably the expressiveness of the agents by introducing strong as well as the default negation. This allows the agents to reason with a *closed or open world assumption*.

Consider the *classical car - train cross* example, where the car wants to cross the rails but it must be *sure* that a train is not coming. We can use the following two modified mental actions to model this situation:

$$\langle \{B(\neg train)\}, cross, \{crossed \leftarrow\}\rangle$$
$$\langle \{B(not\, train) \wedge B(not\, \neg train)\}, listen, \{\neg train \leftarrow \neg sound\}\rangle$$

The first action is of passing the cross when the agent is sure that there is no train coming ($\neg train$). While the second action illustrates the use of the default negation to represent doubt, since the agent will listen when it doesn't know for sure if the train is coming ($not\, train$) or not coming ($not\, \neg train$). This situation was not possible to be modeled in the original 3APL.

From [15], we know that Dynamic Logic Programming is a generalization of Answer Set Programming. Together with the proposition 1, we obtain the following corollary stating that in fact, the agent belief semantics in the modified architecture also generalizes Answer Set Programming.

Corollary 1. *Let $\mathcal{P}$ be an ASP, $x \in \{\cap, \cup, \Omega\}$, and $\langle (\mathcal{P}), \gamma, \Pi\rangle$ be a modified agent configuration. Then:*

$$(\forall L \in \mathcal{L}^{\neg, not}).(\mathcal{P} \models_x L \Leftrightarrow \langle (\mathcal{P}), \gamma, \Pi\rangle \models_x B(L))$$

More Expressive Communications - Agents in 3APL communicate through messages containing only facts. By proposing agents that can communicate programs to other agents, we increasing the possibilities of the multi-agent system. Agents can share knowledge represented by rules. Furthermore, depending on the semantics of the exchanged programs, they could also represent plans or explanations about the environment [4]. As discussed in the previous sections, the agents could update their belief base with theses programs;

Nondeterministic Effect of Actions - As discussed in [3], we can use the multiple stable models of a Generalized Logic Program to represent nondeterministic effects of mental actions. Consider the mental action representing the action of shooting in the famous Yale shooting problem, where the agent tries to kill a turkey with a shot gun, but after shooting, it can happen that the agent misses the turkey:

$$\langle B\,(shoot)\,, shoot, \{kill\_turkey \leftarrow not\, miss;\, miss \leftarrow not\, kill\_turkey\}\rangle\,;$$

There are two possible effects for the action shoot, one if the agent shot the turkey and therefore killed it and another where the agent missed and the turkey is presumably alive.

NP-Complete Complexity - To have the increase in the expressiveness of the language, as investigated in the points above, there is an increase in the complexity of the agent. According to [15] the complexity of computing the stable models is *NP-Complete*.

6 Example

In this Section, we give an example that could be straightforwardly implemented in our modified 3APL system.

Consider the scenario, where *007* is in one of his mission for the *MI6*, to save the world. After infiltrating the enemy base, our special agent encounters the control room where it is possible to deactivate the missile that is threatening to destroy the world as we know it. However, since he was meeting one of the bond girls for dinner, he didn't attend the classes of *Mr. Q* on how to deactivate bombs.

We can represent his belief base as follows:

$$\{\, save\_world \leftarrow \neg bomb \,\}$$

At this point the agent is not able to save the world, since the program has one stable model, namely $\emptyset$. But our agent remembers the briefing of *Mr. Q* before this mission, when *Mr. Q* explained about a special device installed in his watch that could be used to contact the *MI6* headquarters. He immediately takes a look at his watch, presses the special button installed, and asks for further instructions, represented by the communication action, $Send(MI6, request, \{help \leftarrow\})$. The *MI6* headquarters, unable to find *Mr. Q*, sends him some instructions that could be an incorrect one, represented by the following program, P_{MI6}:

$$P_{MI6} : \left\{ \begin{array}{ll} know\_deactivate & \leftarrow not\, wrong\_instructions \\ wrong\_instructions & \leftarrow not\, know\_deactivate \end{array} \right\}$$

Since *007* trusts *MI6*, he updates its beliefs with the modified mental action:

$$\langle B\left(received\left(MI6, inform, P_{MI6}\right)\right), listen, P_{MI6}\rangle\,;$$

With this update, the agent's belief base supports two stable models:

$$\{wrong\_instructions, received(MI6, inform, P_{MI6}\} \text{ and}$$
$$\{know\_deactivate, \neg bomb, save\_world, received(MI6, inform, P_{MI6}\}$$

Notice that the agent must handle the multiple stable models. We consider that for the task of *saving the world* a more conservative approach should be used, namely a *Skeptical* one (where the intersection of all the models is used to determine the agent's beliefs).

Now the spy has to acquire more information about the bomb, since he is not sure if it is possible to deactivate the bomb with the instructions given. If he tries to disable the bomb with the acquired information there can be two outcomes, that the bomb is disabled or that the missile is launched. Represented by the following modified mental action:

$$\langle B\left(not\, know\_deactivate\right), disable\_with\_risk, P_{disable}\rangle$$

where:

$$P_{disable} : \left\{ \begin{array}{ll} \neg bomb & \leftarrow not\, missile\_launched \\ missile\_launched & \leftarrow not\, \neg bomb \end{array} \right\}$$

Therefore, he takes a look at the room (sensing action)[5], and finds the manual of the bomb and realizes that the instructions given were not wrong, updating once more his beliefs with the program:

$$\{\, not\, wrong\_instructions \leftarrow \}$$

With this new knowledge the spy is able to conclude that he knows how to deactivate the bomb (*know_deactivate*), and therefore he is able to disable the bomb ($\neg bomb$), using the following modified mental action:

$$\langle B\,(know\_deactivate)\,, disable\_without\_risk, \{\neg bomb \leftarrow\}\rangle$$

After this action, *007* has safely deactivated the bomb ($\neg bomb$) and finally saved the world (*save_world*) once more (to follow precisely the *007* movies it would be necessary to include somewhere at the end a bond girl...).

In this example we were able to demonstrate several aspects that can be used in the modified 3APL proposed here. First, the use of the strong negation ($\neg bomb$), since it could be incorrect to conclude that the spy saved the world if we used instead default negation (*not bomb*), because there would still be a chance that the bomb is activated but the agent doesn't know it. Second, it was possible to send rules in the communication actions (when the *MI6* headquarters sends 007 the instructions) instead of simple facts. Third, if the agent tried to disable the bomb without the assurance that the information given is correct, there would be a nondeterministic effect after performing the *disable_with_risk* action (bomb being disabled or launching the missile). Finally, we could demonstrate the knowledge evolution, when the agent senses that the instructions were right (*not wrong_instructions* $\leftarrow$), the previous rule (*wrong_instructions* $\leftarrow$ *not know_deactivate*) is rejected and it is finally possible for the agent to save the world (*save_world*).

7 Conclusions

In this paper we proposed a modification to the syntax and semantics of the 3APL language. We investigated the main properties that are obtained by having an agent with a belief base represented by Dynamic Logic Program. The modification proposed considerably increases the expressiveness of the language, by allowing *knowledge updates*, *strong* and *default* negation, more *expressive communication* between the agents. However, to be able to have this expressiveness, there is a clear increase in the complexity of the system.

We investigate in [21], the properties obtained by representing the agent's goal base by a DLP. The agent programmer can elegantly *adopt*, *drop* goals, as well as represent *achievement* and *maintenance* goals. We believe that there would

[5] Notice that we did not deal in this paper with sensing actions, i.e., external actions in the 3APL. However, as the environment is considered as a set of facts, these type of action can be straightforwardly incorporated in our system by updating the agent's beliefs with the sensing information.

be much synergy, if the approaches used here and the approaches in [21] were joined in an unique agent framework.

Even though we believe that the semantics of DLP can handle most of the conflicting cases in an elegant manner, there are some cases that require *program revision*. It would necessary to include a mechanism that would make it possible for the programmer to customize the revision of the programs, for example, by programming the deliberation cycle.

[17] presents a way to represent the social point of view of agents using *Multi Dimensional Dynamic Logic Programs* (MDLP). Further research could be made to try to incorporate these social point views in the 3APL agents, and use this view to decide to consider information sent by another agent or to decide the goals of an agent. A mechanism to update the MDLP would have to be defined, possibly in a similar line as KABUL [14] or MLUPS [16].

References

1. J. J. Alferes, F. Banti, A. Brogi, and J. A. Leite. The refined extension principle for semantics of dynamic logic programming. *Studia Logica*, 79(1):7–32, 2005.
2. J. J. Alferes, J. Leite, L. M. Pereira, H. Przymusinska, and T. Przymusinski. Dynamic updates of non-monotonic knowledge bases. *Journal of Logic Programming*, 45(1-3):43–70, 2000.
3. C. Baral. Reasoning about actions: Non-deterministic effects, constraints, and qualification. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence, IJCAI 95, Montral, Qubec, Canada, August 20-25 1995*, volume 2, pages 2017–2026. Morgan Kaufmann, 1995.
4. C. Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving*. Cambridge University Press, 2003.
5. R. Bordini, J. Hübner, and R. Vieira. Jason and the Golden Fleece of agent-oriented programming. In Bordini et al. [6], chapter 1.
6. R. H. Bordini, M. Dastani, J. Dix, and A. El Fallah Seghrouchni, editors. *Multi-Agent Programming: Languages, Platforms and Applications*. Number 15 in Multiagent Systems, Artificial Societies, and Simulated Organizations. Springer, 2005.
7. R.H. Bordini, L. Braubach, M. Dastani, A. El F. Seghrouchni, J.J. Gomez-Sanz, J. Leite, G. O'Hare, A. Pokahr, and A. Ricci. A survey of programming languages and platforms for multi-agent systems. *Informatica*, 30(1):33–44, 2006.
8. S. Constantini and A. Tocchio. A logic programming language for multi-agent systems. In S. Flesca, S. Greco, N. Leone, and G. Ianni, editors, *Logics in Artificial Intelligence, European Conference, JELIA 2002, Cosenza, Italy, September, 23-26, Proceedings*, volume 2424 of *Lecture Notes in Computer Science*, pages 1–13. Springer, 2002.
9. M. Dastani, M. B. van Riemsdijk, and J.-J. Ch. Meyer. Programming multi-agent systems in 3APL. In *Multi-Agent Programming: Languages, Platforms and Applications*, volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*, chapter 2. Springer, 2005.
10. J. Dix and Y. Zhang. IMPACT: a multi-agent framework with declarative semantics. In Bordini et al. [6], chapter 3.
11. M. Gelfond and V. Lifschitz. Logic programs with classical negation. In Warren and Szeredi, editors, *7th International Conference on Logic Programming*, pages 579–597. MIT Press, 1990.

12. K. Hindriks, F. de Boer, W. van der Hoek, and J.-J. Ch. Meyer. Agent programming in 3apl. *Autonomous Agents and Multi-Agent Systems*, 2(4):357–401, 1999.
13. H. Katsuno and A. O. Mendelzon. On the difference between updating a knowledge base and revising it. In J. A. Allen, R. Fikes, and E. Sandewall, editors, *Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning (KR'91).*, pages 387–394. Morgan Kaufmann, 1991.
14. J. Leite. *Evolving Knowledge Bases*. IOS press, 2003.
15. J. Leite. On some differences between semantics of logic program updates. In C. Lemaître, C. A. Reyes, and J. A. González, editors, *Advances in Artificial Intelligence - IBERAMIA 2004, 9th Ibero-American Conference on AI, Puebla, México, November 22-26, 2004, Proceedings*, volume 3315 of *Lecture Notes in Computer Science*, pages 375–385. Springer, 2004.
16. J. Leite, J. J. A., L. M. Pereira, H. Przymusinska, and T. Przymusinski. A language for multi-dimensional updates. In J. Dix, J. A. Leite, and K. Satoh, editors, *Computational Logic in Multi-Agent Systems: 3rd International Workshop, CLIMA'02, Copenhagen, Denmark, August 1, 2002, Pre-Proceedings*, volume 93 of *Datalogiske Skrifter*, pages 19–34. Roskilde University, 2002.
17. J. Leite, J. J. Alferes, and L. M. Pereira. On the use of multi-dimensional dynamic logic programming to represent societal agents' viewpoints. In P. Brazdil and A. Jorge, editors, *Progress in Artificial Intelligence, Knowledge Extraction, Multi-agent Systems, Logic Programming and Constraint Solving, 10th Portuguese Conference on Artificial Intelligence, EPIA 2001, Porto, Portugal, December 17-20, 2001, Proceedings*, volume 2258 of *Lecture Notes in Computer Science*, pages 276–289. Springer, 2001.
18. J. Leite, J. J. Alferes, and L. M. Pereira. Minerva - a dynamic logic programming agent architecture. In *Intelligent Agents VIII*, volume 2333 of *LNAI*. Springer, 2002.
19. J. Leite and L. M. Pereira. Generalizing updates: From models to programs. In J. Dix, L. M. Pereira, and T. C. Przymusinski, editors, *Logic Programming and Knowledge Representation, Third International Workshop, LPKR '97, Port Jefferson, New York, USA, October 17, 1997, Selected Papers*, volume 1471 of *Lecture Notes in Computer Science*, pages 224–246. Springer, 1998.
20. V. Mascardi, M. Martelli, and L. Sterling. Logic-based specification languages for intelligent software agents. *Theory and Practice of Logic Programming*, 4(4), 2004.
21. V. Nigam and J. Leite. Using dynamic logic programming to obtain agents with declarative goals. In M. Baldoni and U. Endriss, editors, *Pre-Procs. of the 4th International Workshop on Declarative Agent Languages and Technologies, (DALT06), Hakodate, Japan, 2006*, 2006.
22. M. Thielscher. *Reasoning Robots: The Art and Science of Programming Robotic Agents*. Springer, 2005.

Part III

Validation of BDI Agents

Jan Sudeikat[1,2], Lars Braubach[1], Alexander Pokahr[1], Winfried Lamersdorf[1], and Wolfgang Renz[2]

[1] Distributed Systems and Information Systems,
Computer Science Department, University of Hamburg,
Vogt–Kölln–Str. 30, 22527 Hamburg, Germany
Tel.: +49-40-42883-2091
{4sudeika,braubach,pokahr,lamersd}@informatik.uni-hamburg.de
[2] Multimedia Systems Laboratory,
Department of Information and Electrical Engineering
Hamburg University of Applied Sciences,
Berliner Tor 7, 20099 Hamburg, Germany
Tel.: +49-40-42875-8304
{sudeikat,wr}@informatik.haw-hamburg.de

Abstract. Testing and Debugging multi-agent systems (MAS) - which are inherently concurrent and distributed – is a challenging task. While complex application scenarios demand intelligent entities with autonomous reasoning capabilities, the applied reasoning mechanisms impair current approaches to validate MAS implementations. Reactive planning systems, namely the well-known *Belief Desire Intention* (BDI) architecture, have been successfully applied to implement these intelligent entities by means of goal directed agents. Despite testing and debugging, used to validate whether implementations behave as intended, are crucial to serious development efforts, only minor attention has been payed to corresponding tool support and testing procedures for BDI–based MAS. In this paper, we examine how the reasoning mechanism inside agent implementations can be checked and how static analysis of agent declarations can be used to visualize and check the overall communication structure in closed MAS. We present corresponding tool support, which relies on the definition of crosscutting concerns in BDI agents and enables both approaches to the Jadex Agent Platform.

1 Introduction

Agent-orientation proposes autonomous, proactive entities, so-called agents [1], as an atomic design and development metaphor for software systems. These entities enable a lifelike decomposition of software systems as independent actors, interacting with each other. Besides simple reactive agents [2] have been successfully applied in various application domains, the BDI architecture has been established to develop *deliberative* agents [3,4]. Methodologies and development tools are in active development to support the construction of software systems, utilizing this specific architecture. Implementations of this model use the concrete concepts of *beliefs*, *goals* and *plans*, to design and implement individual

R.H. Bordini et al. (Eds.): ProMAS 2006, LNAI 4411, pp. 185–200, 2007.

agents [5,6]. Beliefs denote the local knowledge of individual agents, goals describe the agents objectives and plans are the executable means by which agents achieve their goals. These concepts allow agents to reason pro–actively about which actions to take, i. e. plans to execute.

The autonomous nature of these entities, their complex interactions and their individual memory and reasoning capabilities introduce unprecedented levels of uncertainty [7] to these software systems. While traditional development approaches design the single flow of control in a software system, the individual agent knowledge and reasoning capabilities may lead to unexpected individual behaviors, inhibiting predictions of agent actions and interactions.

Testing and debugging are of equal importance to the efficient development of functionally correct agent systems. While testing (or validation in general) is a more or less systematic approach of discovering unknown bugs, debugging is the process of tracking down and finally removing an already known bug. Much work in the MAS area has been devoted solely to debugging (see e.g. [8,9,10]). This paper discusses approaches, which address validation issues, focusing on approaches for BDI-based agents. Concretely, we present how assertions can be used in BDI concepts to support these activities for BDI agents. In addition, we examine how agent declarations can be used to analyse structural properties of MAS communication and internal agent processing.

This paper is structured as follows. In the coming section we review the BDI architecture. In section 3 testing and validation approaches to MAS are examined and current approaches, particularly concerned with BDI architectures, are discussed. The following section 4 presents our approaches to validate agent implementations. We introduce a mechanism to execute assertions on BDI concepts and two static analysis approaches. After exemplifying their usage and implementation for the *Jadex* system (section 5), we conclude and give prospects for future work.

2 The BDI Agent Architecture

A successful architecture to develop deliberative agents is the BDI model. Bratman [3] developed a theory of human practical reasoning, which describes rational behavior by the notions *Belief*, *Desire* and *Intention*. Implementations of this model replaced the latter two by the concrete concepts *goals* and *plans*, leading to a formal theory and an executable model [4,11].

Beliefs represent the local information of agents about both the environment and its internal state. The structure of the beliefs defines a domain–dependent abstraction of the actual environment. It can be regarded as the *view point* of an agent towards its surrounding. The goals represent agents' desires, which are commonly expressed by certain target states inside the beliefs. This general concept enables pro–active agent behaviors. Agents carry out these goals on their own (see [12] for a discussion of goals in BDI systems). Finally, plans are the executable means by which agents achieve their goals. Agents can access a library of plans and deliberate which plans to execute, in order to reach a desired

target. This mechanism is also known as *reactive planning*, since the precompiled plans are developed at design time. Single plans are not just a sequence of basic actions, but may also dispatch sub-goals.

Both reactive and pro–active behaviors are enabled by internal reasoning processes, composed of *goal deliberation* [13] and *meta-level reasoning* (problems of this are discussed in [14]). The former is the process to select goals to be pursued by an agent, while the latter is responsible to select plans for execution in order to satisfy the previously selected goals. To allow appropriate reasoning, the goals and plans are annotated with *conditions* that describe constraints on their applicability.

3 Validating Multi–agent Systems

In [15] evaluation activities for complex, possibly adaptive and/or distributed software systems have been classified with respect to the amount of expertise required for their application by developers (cf. figure 1). *Testing* refers to the

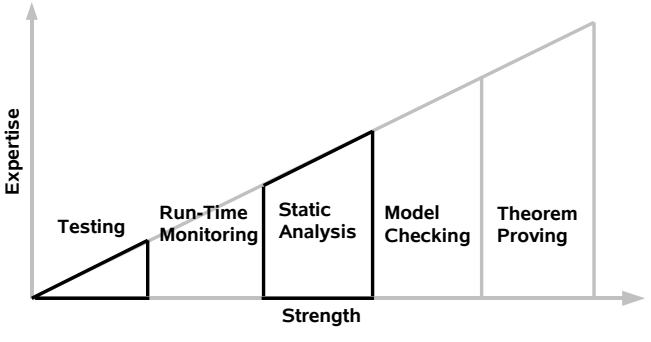


Fig. 1. Categories of evaluation Techniques according to [15]

generation of output by predefined input sequences and comparison of the actual output with expected values. *Run–time monitoring* observes the run–time behavior of software artifacts under given conditions, enabling further analysis. *Static analysis* solely examines the structure of source codes, while *Model Checking* examines all reachable program states in order to verify specifications. In the end, *Theorem Proving* enables formal proofs of correctness (these approaches are extensively discussed wrt. AOSE in [16]). Both Model Checking and Theorem Proving provide most confidence in source codes but require high cost in specification effort and computation as well. While ongoing research [15] is enhancing their applicability, these requirements often impair their application in commercial development settings. These mainly rely on monitoring, static analysis and testing.

We focus here on testing and static analysis, while we address monitoring of BDI agents in [17]. For a generic overview of validating multi–agent systems,

we first focus on communicative aspects (section 3.1), as these can be addressed without considering a specific internal agent architecture. Then we narrow our focus to the peculiarities of the BDI model. We propose a comprehensive validation process that addresses individual as well as multi-agent levels, and give a short review of existing work (section 3.2).

3.1 Validating Agent Communication

The communication between agents is an inherent and foundational property of MAS while the exchanged messages are clearly defined artifacts to be recorded and analysed. As one aspect of communication several approaches exist [18,19] that aim at validating the message exchanges of agents according to the underlying interaction protocols. Interaction protocols are task-specific means to determine strictly all allowable message flows in MAS. Even though it is generally not feasible to verify the accordance to protocols at design time, effective runtime monitoring support can be devised. Typically, runtime monitoring is performed by a tool that observes the message exchanges of agents and engage in actions (e.g. report an error to the application developer) when violations of protocols have been detected.

Additionally, protocol validation message exchanges in a MAS can also be used for analysis and debugging purposes. On the one hand, a visualization of the messages in the system can help to find bugs as irregularities, as agents consuming but not sending messages can be easily identified. On the other hand data mining techniques can be applied to reveal unknown system properties and cluster large amounts of message data in a systematic way [20].

3.2 Validating BDI Concepts

A vision of a comprehensive validation strategy for BDI agents is outlined in figure 2. The validation procedure should move from (1) basic functional modules to (2) the composition of these in individual agents and finally considering the interplay of multiple agents (3) among each other and (4) between the MAS and a surrounding IT infrastructure. Approaches to validate the interplay of the agents commonly do not depend on the internal agent architecture. Therefore, it is expected that approaches as presented in section 3.1 can be used to validate BDI agents as well as non-BDI agents.

Current approaches to validate BDI agents are concerned with (1) the compliance of agent execution to design artifacts drawn from development methodologies [18], (2) the comprehension of agent behaviors by comparing models of the expected order of reasoning events [21,22] and (3) test case generation for BDI–plans, based on coverage criteria [23].

While Padgham et al. use design artifacts from the *Prometheus* methodology [24] mainly for validation of communication protocols [25,18] and agent communication [26], they also define and test for *coverage* and *overlap* of BDI plans. BDI reasoning events have full coverage when the event is expected to

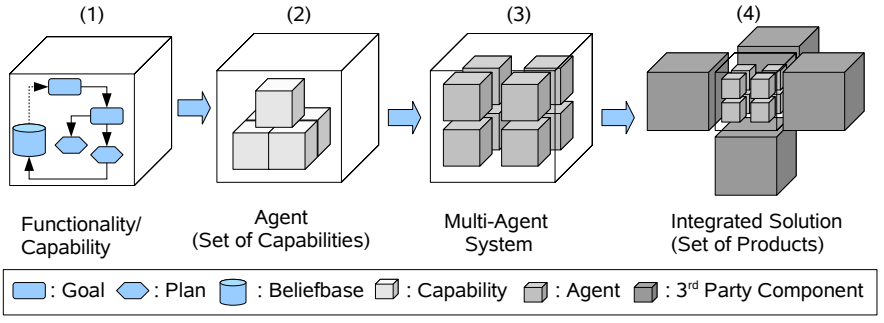


Fig. 2. Typical testing stages for a MAS. The System is developed from single functionalities to an integrated solution which interfaces 3rd party software packages.

have an applicable plan under any condition and overlap describes that multiple plans may be applicable to handle an event. In [18] these criteria are validated by automated introduction of logging code to monitor plan adoption.

In [22] Lam et al. propose a method and a tool for comprehending and explaining agent behavior. In principle, the method is an extension to the runtime monitoring of message exchanges, by considering internal reasoning activities of the agents as well. All these behavioural data is collected at runtime in a knowledge base and can be used for interpretation purposes e.g. to understand why an agent has executed a specific action. The tool – *Tracer* – also allows to view the order of reasoning events in graph structures, facilitating error detection by exploiting pattern recognition.

In [23] an automatic BDI test-case generation method has been proposed. The method is able to create test cases in accordance with coverage criteria. These coverage criteria are plan as well as node (plan statement) based and ensure that test cases for all relevant execution paths of the agent will be generated. A platform independent prototype tool – *BDITester* – implements the concepts and can be used to produce a set of test cases. Even though the tool includes a mechanism to calculate and reduce the necessary number of test cases, it remains unclear if the approach scales up for complex systems and if domain related errors can be detected.

4 A Practical Approach to the Validation of BDI Reasoning

In the case of BDI agents in general, proper functioning is based on the processed BDI concepts. These comprise (1) belief consistency, (2) proper goal adoption and consistency and finally (3) correct plan execution. Agent developers declare the BDI concepts and annotate conditions to their applicability in order to define the behavior of the goal directed agents. Since these agents reason pro–actively

about their goal and plan adoption, verification of agent *reasoning mechanisms* is crucial and challenging. While functional properties only ensure that subsequent agent actions are executed properly, a comprehensive testing procedure needs to assure that agents will come to the intended conclusions, i. e. adopt appropriate goals and plans.

In order to address these issues on BDI agent validation, we present a novel testing and validation approach for BDI agents. It comprises two parts: To test agents and identify misconceptions in agent code, we propose the contributive execution of assertion statements, triggered by BDI reasoning events. Secondly, we discuss how a static analysis of BDI agent declarations can improve the consistency of agent specifications and multi-agent interplay.

4.1 Assertions in BDI-Concepts

Assertions are typically provided as extensions to programming languages[1]. After we briefly introduce assertions in general, we classify which properties in BDI reasoning can be validated by using them in BDI agents.

Assertions in Software Engineering. Following Hoare [27], an assertion is:

> "... a Boolean formula written in the text of a program, at a place where its evaluation will always be true or at least, that is the intention of the programmer..."

If an assertion statement evaluates to false, the program has entered an inconsistent state. Assertions have their origin in program verification [28] and can be traced back to the founding works of Turing [29], who introduced this concept to specify *interfaces* between parts of programs. Despite their age, assertions are widely used in the software industry. The *design by contract* principle [30] is closely related to object–oriented development and assertions lend themselves to detect, diagnose and classify violations of these contracts specified as *pre–* and *post–conditions* (e.g. in the *Eiffel* programming language).

Defects in programs can only be identified when testing efforts lead to observable incorrect output. This *observability* of software artifacts requires that (1) an input causes a defective code to be executed, (2) program data gets corrupted and finally (3) this corrupted data is propagated to an incorrect output [31]. Components are usually tested using *unit–test*[2] frameworks, which facilitate instantiation, automated method calls and comparison of return values to expected output. Though these tests can usually be used to examine corrupted object states, *encapsulation* and *information hiding* may mask errors in *integration* and *system–level* tests, which are used to examine the interplay of components and subsystems. To complete testing approaches, assertions have been proposed to increase the probability that incorrect outputs occur when erroneous code is executed [31].

[1] e. g. introduced to Java in version 1.4.
[2] e. g. http://www.junit.org

Application of Assertions in BDI Concepts. As described in section 3.2, the validation of the BDI–based reasoning process is a major challenge in testing and debugging BDI agents. While only message exchange and external agent actions are observable on the MAS level, it is necessary for developers to be confident that the intended goals and plans are adopted during agent execution. While encapsulation and information hiding may be detrimental to state–error propagation in object–oriented systems, the same is true for the event–based and condition centric reasoning cycles in BDI agents.

In this respect assertions can be used to (1) specify and ensure the *relations* between BDI concepts and the surrounding agent as well as, (2) *invariant* properties in agent/MAS execution. Whereas the conditions which are annotated to BDI goals and plans enable automated reasoning, developers intend specific agent configurations and behaviors. Assertions can be used to annotate explicit specifications of the intended context, e. g. agent configurations, to BDI concepts. These annotations supplement concept properties by automated notification of violations to increase the observability of unintended and/or inconsistent agent states. According to the well established *design by contract* principle [30] in object–oriented systems, similar contracts – between agent states and BDI concepts – can be specified on the BDI–concept level. In opposition to programming language assertions, execution of BDI–based assertions is triggered by BDI reasoning events. The statements are used to ensure that BDI events occure in intended agent configurations.

As there is no general consensus how the BDI concepts should be represented at the implementation level, different kinds of representations may require different kinds of assertions. For beliefs at least two different representational forms have been used. Logic based representations such as first order predicate logic could e.g. use assertions about which facts are allowed/prohibited to be contained in the beliefbase and also which relationships may occur. When relational and object-oriented representations are employed assertions could e.g. be used to restrict the allowed belief values (define the domains).

The same representational variety applies for goals which range from purely procedural goal events to strictly declaratively specified goals. In general holds, that the more information about a goal can be supplied the more data is available for assertion evaluation. One interesting aspect for testing concerns the creation of goals if this aspect is covered by the concrete architecture. Invariants may be specified by assertions in order to highlight when unintended goals are instantiated in specific agent states.

The most conceptual similarity among BDI concepts exists in the area of plans. Even though there is no standard way how and in which language a plan body should be realized, the key elements of plan heads are generally accepted (although alternative terms for the same element exist). This allows to defer generic properties that can be used to validate plan reasoning. Most notably, the coverage and overlap criteria, have been proposed by [26] to allow runtime monitoring of event resp. goal processing. Assertions could be used to capture the semantics of coverage and overlap criteria (cf. section 3.2).

The annotated conditions also provide additional documentation for the agent code. In section 5 we present an implementations that executes assertions at every state change of the annotated element. Developers need to be aware of drawbacks. While the execution of assert–statements is intrusive to the agent execution, extensive processing in these statements will slow down the agents and side effects in assertion statements may impair proper agent execution.

4.2 Static Analysis

BDI–based MAS are composed of a set of agent declarations which define the properties of BDI concepts and further implementation dependent details. Figure 3 gives a canonical overview of such a MAS, composed of n agent types. The declared agent properties comprise the *messages* to be sent and received as well as *internal events* which may trigger plan execution (e. g. found in [4,32,33]). The consistency of declarations of these important implementation concepts can be checked in order to validate structural properties and highlight misconceptions.

The Static Structure of Internal–Events. Internal events typically trigger plan adoption as a mean of intra–agent communication (cf. figure 3). Therefore, proper specification of these events and their triggering function can be analysed by iteration and comparison of event declarations. As these events can also be directly handled in plans, one has to require that developers declare this behavior, e. g. via *meta data* in plan implementations, in order to validate that all specified events actually trigger plans and vice versa all triggers are correctly declared.

Static Analysis of Message–Events. As outlined in figure 3 agent declarations comprise the messages to be sent and received. Therefore a set of agent declarations can be understood as a graph G, which is defined as a tuple $G = \langle A, M, Ae, Me \rangle$, where A denotes a set of agents, M a set of messages and Ae a set of edges between agents and messages ($Ae \subseteq A \times M$) and finally Me denotes a set of edges between sent and received messages ($Me \subseteq M \times M$). Since message declarations comprise implementation details (e. g. FIPA[3] compliant performatives, utilized ontologies, etc.) used to control sending and reception, the edges between messages (Me) describe matching pairs of sent and received messages ($(m, m') \in Me \Leftrightarrow m, m' \in M$)[4]. Identification of these is therefore dependent on the applied agent platform. In order to identify design flaws we expect that all messages (m) declared in one agent (a) can be sent/received (m') by at least one other agent (a') in a MAS:

$$\forall (a, m) \in Ae.\ \exists (a', m') \in Ae \wedge (((m, m') \vee (m', m)) \in Me)$$

[3] http://www.fipa.org/

[4] When messages are declared globally, e. g. in the JACK agent platform [32], Me becomes the identity relation.

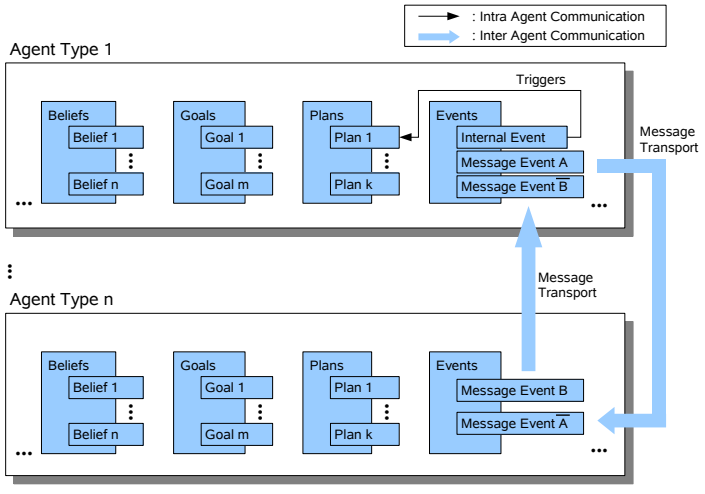


Fig. 3. A canonical view on a Jadex–MAS. One to n agent types are declared. Among the properties of BDI concepts and implementation details, *Internal* and *Message Events* are specified. Only matching message events, e. g. A and $\bar{A}$, enable communication.

Message–based communication is an essential property of MAS, that can be represented in a gaph structure, suitable to verify static MAS properties. We expect that agent declarations in other agent platforms and design methodologies can be exploited to define similar graph structures, describing possible message exchanges as well as other MAS properties, e. g. shared resources or offered services.

5 A Case Study – The Marsworld in Jadex

To exemplify the usage of the above described testing and analysis tools, we examine an example MAS from the Jadex–Project. This example scenario has been inspired by a case study in [34], where hierarchical structures of static, predefined roles are examined. In order to allow for cooperative behavior, the system has been generalized as follows. The objective for a group of robots (agents) in the so–called *Marsworld*, is to mine ore on a far distant planet. The mining process is composed of (1) *locating* the ore, (2) *mining* it on the planets surface and (3) *transporting* the mined ore to the home base. Therefore, a collection of three distinct types of agents are released from a home base to a bounded environment. All of them have a sensor range to detect occurrences of ore in the soil and start immediately a searching behavior. Sensed occurrences of ore are reported to the so-called *sentry*-agent. This robot is equipped with a wider sensor range and can verify, whether a suspicious spot actually accommodates ore or not. When ore is found, the location is forwarded to a randomly selected *production*-agent,

equipped with a dedicated mining device. After mining is finished, a group of *carry*-agents is ordered to transport ore to the home base. When the ordered actions have been performed agents continue searching. Details on the dynamics of this MAS can be found in [17].

5.1 Validation Support for the Jadex System

The Jadex reasoning engine [33,35] provides an execution environment for BDI-style agents on top of arbitrary distributed systems middleware. The individual agents consist of two parts. First, each agent is described by a so–called *Agent Description File* (ADF), which denotes the structure of beliefs, goals and plans, events, and sent and received messages in XML syntax. Secondly, the activities an agent can perform are coded in plans, which are ordinary Java classes. Plan descriptions in the ADF (so–called *heads*) reference the compiled Java classes (so–called *body*) and denote the conditions, events, or goals, which may lead to plan instantiation.

Jadex already does integrity checks of the XML agent descriptors, but these are only syntax and type checks. In addition, the Jadex tool suite comprises a a so–called *Test–Center* for the automated execution of *unit–tests* at the agent level. Inspired by the successful *JUnit* framework for object–oriented software systems, the Test-Center facilitates the execution of test *suites* composed of test *cases*. The latter ones are encapsulated by dedicated test agents. E. g. this allows to test capabilities that encapsulate specific application logic. In this respect, it is still an open question how agent interferences can be avoided that are caused by pro-active agent behaviors.

The work presented here focuses on semantic checking of agent systems. For application specific semantic checks at runtime, we introduce the assertion concept for the agent specifications. Annotating an assertion to an element of the agent, such as a belief, goal, or plan allows the system to generate immediately detailed error messages, whenever the system does not behave as originally intended by the agent developer. Moreover, the XML representation of Jadex agents lends itself to easy static analyses. We show how the internal and message event issues from section 4.2 can be analyzed, and present a tool, which visualizes the communication structure of a multi-agent system in a way, that makes it easy to spot specification errors, directly from the visualization.

5.2 Checking Consistency Using Assertions

The implemented assertion mechanism executes arbitrary Java statements, that can be annotated to beliefs (including beliefsets), goals and plans in `assertion` tags in agent ADFs. The annotated statements will only be executed when the ADF comprises a reference to the capability `jadex.assertion.Assert`, therefore simple ADF modification allows to turn assertion execution on and off. Assertion statements are expected to evaluate to `true`. When they are violated a detailed warning will be generated, specifying the agent and the element where the assertion evaluated to `false`.

Figure 4 exemplifies the usage of assertions. This code fragment is taken from the *sentry* agent of the marsworld example, which stores reported and found ore locations in a beliefset named *my_targets*. The shown code checks whether the sum of stored values does not exceed the amount of targets in the game environment (defined in a class `Environment`), which can be accessed from within the beliefs of the agent.

```
<!-- The seen targets. -->
<beliefset name="my_targets" class="Target">
        <assertion>
                $agent.getBeliefbase().getBeliefSet("my_targets")
                .getFacts().length
                        &lt;=
                ((Environment) $agent.getBeliefbase().getBelief("environment")
                .getFact()).getTargets().length
        </assertion>
</beliefset>
```

Fig. 4. An assertion statement added to a *beliefset* description in a jadex ADF. The statement checks the maximum amount of a target set. The < entity is used to escape the less than (<) character.

The `Assert` capability implements a listening object to all events originated from belief access, goal or plan adoption. For all state changes of these elements this listening object looks up annotated assertion statements and executes them. This mechanism has been implemented as a *crosscutting concern* in the Jadex system.

Crosscutting Concerns in BDI–Agents. Aiming towards automated assertion execution, we utilized an enhancement to this modularization concept, which allows to define *crosscutting concerns* in agent implementations. In [14] so–called *capabilities* have been proposed to modularize BDI agents. These capabilities comprise beliefs, goals, plans and a set of visibility rules of these elements to the surrounding agent. In development of MAS, they are used to define specific functionalities which can be imported by different agent types.

Modularization is a crucial concept in software engineering, following the *Separation of Concerns* principle. The functionality of a software system can be decomposed into *core concerns*, which are to be separated into different components or modules [36] and so–called *aspects* which crosscut them [37]. Crosscutting prime examples are inter alia failure recovery and logging.

In this respect capabilities [38,14] intend to define and modularize core concerns in BDI agents. Agent types can share functionality by inclusion of the same capability. Similar to conventional development efforts — without the notion of aspects — non–functional concerns can be captured in modules and executed by explicit references to elements inside these modules. So–called *co–efficient capabilities* (CC) automate this referencing by exploitation of the local reasoning mechanisms. We name these capabilities *co–efficient*, because they register for *contributive* processing on certain BDI reasoning events. These occur during

agent execution and cause further agent actions, e. g. belief changes, goal adoptions and plan instantiation. While the agent executes as specified, additional processing can be triggered. Whereas it is possible to to modify the surrounding agent, this mechanism allows crosscutting functionalities, like logging, failure recovery etc., to be automatically triggered, without explicit references in goals or plans. Details of their implementation, a discussion of similarities and differences to object–oriented aspects and their usage for minimum intrusive plan observation can be found in [17].

Crosscutting Assertion Execution. Finally, we outline a prototype implementation that allows the crosscutting execution of assertion statements in BDI agents. While assertion statements are annotated to beliefs goals and plans, the execution mechanisms has to ensure that these statements are evaluated whenever these elements are accessed by the BDI agent interpreter. Therefore, the crosscutting implementation is encapsulated in a *co–efficient* capability, which is registered to react to all BDI reasoning events related to the annotated elements.

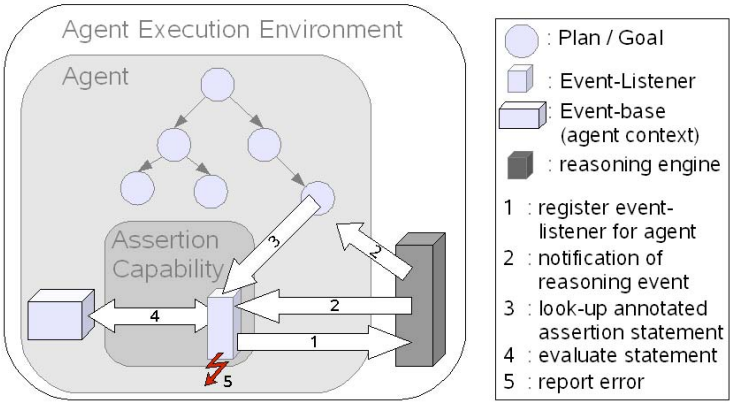


Fig. 5. Assertion Execution in BDI agents

The execution cycle of the corresponding implementation is outlined in figure 5. On agent start–up the capability triggers the registration of an event–listener for the surrounding agent (1). This listener is notified when corresponding BDI–reasoning events occur in the reasoning engine that controls the event–based execution cycle [4] of BDI agents (2), looks–up annotated assertion statements (3) and executes these (4) in the agent context. When this Java statement does not evaluate to true an appropriate error message is generated (5).

5.3 Internal Event Consistency

A typical declaration of an internal event (`examination_finished`) is exemplified in figure 6, taken from the sentry agent of the marsworld example.

```
<plans>
    <plan name="call_producer">
        <body>new CallProducerPlan()</body>
        <trigger><internalevent ref="examination_finished" /></trigger>
    </plan>
</plans>
<events><internalevent name="examination_finished" /></events>
```

Fig. 6. An internal event triggers the execution of a plan

Dispatching this event triggers the execution of a plan called `CallProducerPlan()` to search for an available producer agent to mine ore. Declared InternalEvents can be dispatched within plans via the Jadex API. We implemented a capability (`jadex.iecheck.IECheck`) which checks whether all declared events trigger plans when the agent starts. Although internal events typically trigger plans in the presented way, it is also possible that plans handle these events directly by a blocking call of `waitForInternalEvent(String type, long timeout)` or `waitForInternalEvent(String type)`. Therefore our implementation utilizes the novel annotation mechanism of Java 5.0[5] to handle these cases. Developers are expected to annotate the handled events to the plan classes using this meta–data facility.

5.4 MessageEvent Consistency

Figure 7 exemplifies the declaration of a message event. The used performative, transmission language and ontology are declared. In addition, the direction of the message need to be specified. While this example is declared to be sent (direction=`send`), possible values are also `receive` and `send_receive`. We developed a tool to examine the declared messages in a set of ADFs belonging to an application. Message properties and declared directions are compared in order to compute possible message exchanges. These are reported together with orphaned message events, where no matching sender/receiver is specified.

The found message matches are displayed in a graph structure (according to 4.2) as exemplified in figure 8. In these graphs agents (bigger, light) and message events (smaller, dark) are denoted as nodes. Messages are connected to the declaring agent via *aggregation* edges (following UML[6] notation), while possible message exchanges are represented by dark arrows. Since all messages are displayed, i. e. are not structured in the protocols involved, we display the MAS in a three dimensional space to allow efficient layout. Graph representations are generated to be displayed with the *Wilmascope*[7] tool. This tool visualizes XML representations of graphs and allows users to set various rendering options to control a force directed layout. Therefore users can adjust the graph layout according to their needs and examine the graph closely in a virtual space.

[5] http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html
[6] http://www.omg.org
[7] http://www.wilmascope.org/

```
<events>
    <!-- Call producer agent to mine ore at the given location. -->
    <messageevent name="request_producer" type="fipa" direction="send">
        <parameter name="performative" class="String" direction="fixed">
            <value>SFipa.REQUEST</value></parameter>
        <parameter name="language" class="String" direction="fixed">
            <value>SFipa.JAVA_XML</value></parameter>
        <parameter name="ontology" class="String" direction="fixed">
            <value>MarsOntology.ONTOLOGY_NAME</value></parameter>
    </messageevent>
</events>
```

Fig. 7. Declaration of a message event in the *sentry* agent of the *marsworld* example. This message transmits a location to the *production* agent to order the mining of ore.

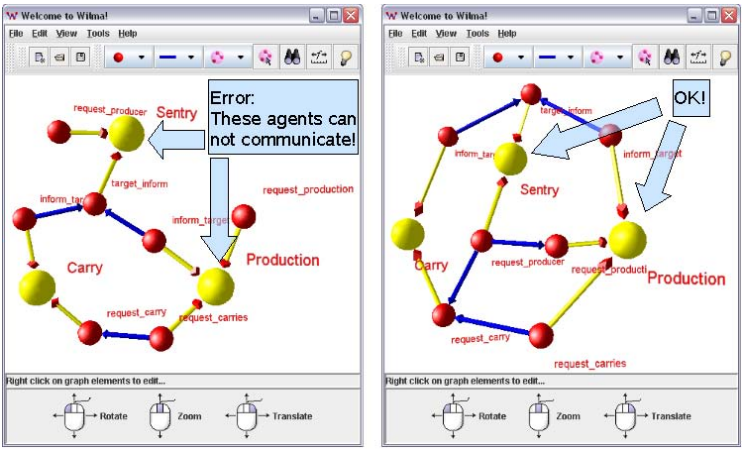


Fig. 8. A defective (left) and correct (right) communication–network of the Marsworld example. Bigger vertices (light colored) denote Agents while the smaller ones (dark colored) represent MessageEvents in the agent ADFs.

Figure 8 (right hand side) displays the static communication structure of the marsworld example. *Carry* and *production* agents can report ore locations (`inform_target`) to the sentry agent. Sentry agents can order mining by the *production* agent (via `request_producer`), and *producer* agents in turn can order the transportation of ore by carry agents (`request_carry`). The left hand side shows the same MAS with a mistake in the declaration of the `request_producer` message. The force directed layout highlights that sentry and producer agents can not correctly communicate as message events are orphaned.

6 Conclusions

In this paper we have highlighted that the validation of agent behaviors is a major challenge in MAS development. In order to test BDI agent implementations,

i. e. agent behaviors based on BDI reasoning processes, we proposed assertions, exemplified their usage and outlined a crosscutting implementation. Assertion statements are annotated to BDI elements and their execution is triggered by BDI reasoning events. In addition, we discussed the static analysis of agent declarations to improve the overall consistency of agent implementations and to detect specification errors. A prototype enables to verify the consistency of declared internal events and messages. Finally, the overall communication structure of MAS can be visualized as three dimensional graphs.

The presented visualization approach is still preliminary. We plan to enhance display and user interaction to show dynamic properties of agent execution (cf. [17]). The adoption of assertions in actual development efforts will reveal benefits and limitations of the proposed validation approach. Examination of common bugs and debugging strategies for BDI agents may inspire the methodical usage of assertions, i. e. a structured process to derive assertions from agent declarations. In addition it needs to be taken care that violations of these are in fact reflecting inconsistent agent states and the assertions have no side effects. While we proposed here the usage of assertions to check individual agent reasoning, it is an open research question whether assertions on the MAS level can be inferred that describe invariant properties on the agent interplay.

References

1. Odell, J.: Objects and agents compared. Journal of Object Technology **1** (2002)
2. Brooks, R.A.: Elephants don't play chess. Robotics and Auton. Sys. **6** (1990) 3–15
3. Bratman, M.: Intentions, Plans, and Practical Reason. Harvard Univ. Press. (1987)
4. Rao, A.S., Georgeff, M.P.: BDI-agents: from theory to practice. In: Proceedings of the First Intl. Conference on Multiagent Systems. (1995)
5. Georgeff, M.P., Lansky, A.L.: Reactive reasoning and planning: an experiment with a mobile robot. In: Proc. of AAAI 87, Seattle, Washington (1987) 677–682
6. Pokahr, A., Braubach, L., Lamersdorf, W.: A flexible BDI architecture supporting extensibility. In: The 2005 IEEE/WIC/ACM Int. Conf. on IAT-2005. (2005)
7. Jennings, N.R.: Building complex, distributed systems: the case for an agent-based approach. Comms. of the ACM **44 (4)** (2001) 35–41
8. Liedekerke, M.H.V., Avouris, N.M.: Debugging multi-agent systems. Information and Software Technology Journal **37** (1995) 103–112
9. Ndumu, D.T., Nwana, H.S., Lee, L.C., Collis, J.C.: Visualising and debugging distributed multi-agent systems. In: Proc. of AGENTS '99. (1999) 326–333
10. Flater, D.W.: Debugging agent interactions: a case study. In: Proceedings of the 2001 ACM Symposium on Applied Computing (SAC), ACM (2001) 107–114
11. Rao, A.S.: Agentspeak(l): Bdi agents speak out in a logical computable language. In: MAAMAW '96: Proceedings of the 7th European workshop on Modelling autonomous agents in a multi-agent world. (1996) 42–55
12. Braubach, L., Pokahr, A., Lamersdorf, W., Moldt, D.: Goal representation for BDI agent systems. In: Proc. of PROMAS'04. (2004)
13. Pokahr, A., Braubach, L., Lamersdorf, W.: A bdi architecture for goal deliberation. In: Proc. of AAMAS '05. (2005) 1295–1296
14. Busetta, P., Howden, N., Rönnquist, R., Hodgson, A.: Structuring BDI agents in functional clusters. In: ATAL '99, Springer-Verlag (2000) 277–289

15. Menzies, T., Pecheur, C.: Verification and validation and artificial intelligence. In Zelkowitz, M., ed.: Advances in Computers. Volume 65., Elsevier (2005)
16. Timm, I.J., Scholz, T., Frstenau, H.: IV From Testing to Theorem Proving. In: Multiagent Systems - Intelligent Applications and Flexible Solutions. to be published by Springer (2006)
17. Sudeikat, J., Renz, W.: Monitoring group behavior in goal–directed agents using co–efficient plan observation. In: Proc. of the 7th Inernational Workshop on Agent Oriented Software Engineering (AOSE'06). (2006)
18. Padgham, L., Winikoff, M., Poutakidis, D.: Adding debugging support to the prometheus methodology. Engin. Applications of Art. Intel. **18** (2005) 173–190
19. Chesani, F.: Formalization and verification of interaction protocols. In: ICLP. (2005) 437–438
20. Botía, J.A., López-Acosta, A., Gómez-Skarmeta, A.F.: ACLAnalyser: A tool for debugging multi-agent systems. In: ECAI. (2004) 967–968
21. Lam, D.N., Barber, K.S.: Automated interpretation of agent behavior. In: Workshop for Agent-Oriented Information Systems (AOIS-2005). (2005)
22. Lam, D.N., Barber, K.S.: Comprehending agent software. In: Proc. of the 4th int. joint conf. on autonomous agents and multiagent systems (AAMAS '05). (2005)
23. Low, C.K., Chen, T.Y., Rönnquist, R.: Automated test case generation for bdi agents. Autonomous Agents and Multi-Agent Systems **2** (1999) 311–332
24. Padgham, L., Winikoff, M.: Developing Intelligent Agent Systems: A Practical Guide. Number ISBN 0-470-86120-7. John Wiley and Sons (2004)
25. Poutakidis, D., Padgham, L., Winikoff, M.: Debugging multi-agent systems using design artifacts: the case of interaction protocols. In: Proc. of AAMAS '02. (2002)
26. Poutakidis, D., Padgham, L., Winikoff, M.: An exploration of bugs and debugging in multi-agent systems. In: Proc. of ISMIS 2003. (2003)
27. Hoare, C.A.R.: Assertions: a personal perspective. Software pioneers: contributions to software engineering (2002) 356–366
28. Floyd, R.: Assigning meaning to programs. Mathematical Aspects of Computer Science **XIX American Mathematical Society** (1967) 19–32
29. Turing, A.M.: Checking a large routine. In: Report on a Conference on High Speed Automatic Calculating Machines, Cambridge University Mathematical Lab. (1949)
30. Meyer, B.: Object Oriented Software Construction. Prentice Hall (1997)
31. Voas, J.: How assertions can increase test effectiveness. IEEE Software **March/April** (1997) 118–122
32. Busetta, P., Rönnquist, R., Hodgson, A., Lucas, A.: Jack - intelligent agents – components for intelligent agents in java. Technical report, Agent Oriented Software Pty. Ltd (1998)
33. Pokahr, A., Braubach, L., Lamersdorf, W.: Jadex: A bdi reasoning engine. In: Multi-Agent Programming. (2005) 149–174 Book chapter.
34. Ferber, J.: Multi-Agent Systems. Addison Wesley (1999)
35. Braubach, L., Pokahr, A., Lamersdorf, W.: Jadex: A BDI Agent System Combining Middleware and Reasoning. In: Software Agent-Based Applications, Platforms and Development Kits, Birkhäuser (2005)
36. Parnas, D.L.: On the criteria to be used in decomposing systems into modules. Commun. ACM **15** (1972) 1053–1058
37. Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In: Proc. of ECOOP. Springer (1997)
38. Braubach, L., Pokahr, A., Lamersdorf, W.: Extending the capability concept for flexible BDI agent modularization. In: Proc. of PROMAS-2005. (2005)

A Tool Architecture to Verify Properties of Multiagent System at Runtime

Denis Meron[1] and Bruno Mermet[2]

[1] LITIS, University of Le Havre, France
[2] GREYC, University of Caen, France
Denis.Meron@univ-lehavre.fr
Bruno.Mermet@univ-lehavre.fr

Abstract. This paper describes an architecture allowing to verify properties of a multiagent system during its execution. This architecture is the basis of our study whose goal is to check at runtime, if agents and more generally multiagent systems satisfy requirements. Considering that a correct system is a system verifying the properties specified by the designer, we are interested in the "property" notion. That is why we give here a definition of "property" and we present an architecture to validate them. The architecture, a multiagent system itself, is based on a set of agents whose goals are to check at runtime the whole system's properties. So after a brief description of the "property" notion, we describe our architecture and the way to check systems.

1 Introduction

Today, a huge number of scientific areas use computers either to solve problems or to design simulations. Indeed, problems become more and more complex and it is asked to computer science to study and solve such problems. As a consequence, multiagent systems (MAS thereafter) are considered as a solution to take this complexity into account, and are more and more used to design new softwares. However, MAS development is a complex process because many autonomous entities evolve concurrently and asynchronously. So, as it is performed for softwares developed using the object paradigm, multi-agent programming needs to validate and verify MAS developed. In other words, when a MAS is developed, it is required to check whether the system has been correctly developed.

This idea is not new and several research have already been performed on verification and validation of MAS [6,20,4,18,10]. Among them, few ones deal with the way to prove the correctness of MAS either by theorem proving [20] or by model-checking [1]. However, the proof may be difficult to perform especially when the problem is massively distributed with many interactions between agents. Furthermore, model-checking costs time and the problem must be reduced to a finite one, with a limited number of states (even if unbounded model-checking using binary decision diagrams reduces this limitation). A second way to validate softwares is to perform tests, thanks to an architecture making testing feasible. Contrary to the model-checking, tests are not required to be exhaustive. As a consequence, the confidence brought by tests is reduced, but is easier to obtain. But in the case of MAS, which are open systems, the coverage of test is difficult to

R.H. Bordini et al. (Eds.): ProMAS 2006, LNAI 4411, pp. 201–216, 2007.

assess and is often near zero. Moreover, because of the asynchronous execution schema of the agent, a given test scenario is not guaranteed to give the same result twice.

In this context, in order to check that a MAS was correctly developed, it is useful to design an architecture allowing to test the final system on the fly. Indeed, we need to design a set of agents whose goal is to detect and highlight any effective behavior that is not compatible with the expected behavior. Assuming that a well implemented MAS has the properties that the designers implemented as a principle, we think that checking a system must rely on the verification of these properties. Moreover, to validate a large scale of MAS, we think that the architecture needs to be independent of any system or agent oriented framework. The rest of this paper is structured as follows: section 2 presents some existing studies on system validation. Section 3 defines the notion of property. In section 4, the architecture we designed to check properties at runtime is exposed. Finally, Section 5 concludes the paper, with an indication of on-going work.

2 State of the Art

Since the birth of computer science, when designing software for critical systems (where life may depend on it) or just toys problems, the necessity to guaranty both the correctness of the design and the correctness of the system execution appeared as a major problem for developers. But most of agent oriented framework like JADE, AgentBuilder, Jack or MADKIT, don't give really issue for debugging [6]. So it appears that it is necessary to design an architecture allowing to check the integrity of the system. The idea is not new and several researches have already be done by Steven SAPHIRO [20], David FLATER [4], or also David POUTAKIDIS [18,19]. In the three next sections we will see the different ways they use to check systems.

2.1 Specification Language

Steven SHAPIRO *et al.* in [20] describe a language to specify MAS able to prove that a system was correctly designed. The language named CASL for Cognitive Agents Specification Language is a framework which has a mix of declarative and procedural components to facilitate the specification and the verification of multiagent systems. The main idea is to specify the mental states of an agent in a mathematical way, so it may be provable. More particularly the language allow to specify the agent information (such as knowledges and beliefs) and its goals. They have also developed a verification environment (CASLve) for their language based on the Prototype Verification System (PVS) [17] to make it easier to verify properties of CASL specification. The verification environment provides the user a comprehensive library of proof methods, or lemmas.

This method is very interesting but the proof of a complex system is very difficult to obtain. Moreover, they are not interested in problems that may appear at run time or by the effective agent behavior.

2.2 Behavior Study

Some studies rely on collecting information that represent the behavior of the system to control [21,16]. The focus in such debugging tools is on collection of information

(message exchange, interaction, etc) and on the presentation to the user with filtering applied to the message. However, as David FLATER writes in [4], the study of such views of MAS is long and difficult. Furthermore these methods have some limitations [18]:

- too many information messages are presented to programmers, making difficult to understand what is really happening in the system.
- Without a proper procedure for identifying what sorts of information to look for, it is unrealistic to know in advance what information will be useful when trying to debug the system.
- Most systems have no means of identifying where problems may occur, even if the developer notices an error, it could take an unnecessarily long time to pinpoint the location of this error.
- They rely on the right interpretation of the information by the programmer. Since the output of most of the debugging tools is raw messages the developer needs to inspect the content of the messages and the flow of messages and try to determine what is going wrong. With a large number of messages, this can be extremely difficult.

In their study David POUTAKIDIS *et al.* [18,19], describe a new way for validate the system. They use a two stages methodology. First they study the system during the design stage to collect informations on the agents. Secondly they integrate these informations on an agent witch goal is to validate the system behaviors.

« Our central thesis is that the design documents and system models developed when following an agent-based software engineering methodology can be incorporated in an agent and used at run-time to provide for run-time error detection and debugging. »

In this paper they are interested by the case of communication exchange. During the design stage they translate all possible conversations into Petri Nets. The debugging agent has a library of known protocols (represented as Petri nets). When a new conversation is started (i.e. the debugging agent received a message which isn't part of an existing conversation) the debugging agent instantiates all protocols which are capable of beginning with the received message. If a message received is not compatible with a known conversation, the agent send an error signal.

This way to check the system is very interesting and our research takes inspiration from this work but there is also some limitation. Indeed the conversation translation in Petri Nets may very fastidious if there are a lot of agents whose communicate. Furthermore to check the conversation, all the agents must send a carbon copy to the debugging agent. This task, and the authors are aware of this, may dramatically disturb the system execution particularly if the system is time dependent.

2.3 Exception Handling

Another way to check systems is given by Mark KLEIN *et al.* [10]. The main idea is to add agents to the system whose goals are to spot any departure from the « ideal collaborative behavior ». All of these departures are called exceptions. This approach is taking inspired by decision support systems such as those used in medicine where a description of symptoms leads to a diagnostic.

« This service can be viewed as a kind of "coordination doctor"; it knows about the different ways multi-agent systems can get "sick", actively looks system-wide for symptoms of such "illnesses", and prescribes specific interventions instantiated for this particular instance from a body of general treatment procedures. »

However the doctor agents need to have a knowledge base of generic exception handling detection, diagnosis and resolution expertise. This data base may be very important to feet with a large scale of systems' exceptions. So weakness and portability of the framework may be compromise.

2.4 Design by Contract

Bertrand MEYER in [14,15] described a software engineering theory based on contracts. The central idea is that software entities have obligation to other entities based upon formalized rules (assertions) between them. The two entities rely by contract are called caller and routine. Caller must guarantee certain conditions before calling the routine (precondition) and the routine guarantee certain properties after the call (postcondition). Precondition binds the caller and postcondition binds the routine. Beside there is also an assertion rely on invariant that must be satisfied by every instance of the class.

The language that offers the best support for design by contract is Eiffel, designed by Bertrand MEYER (www.eiffel.com), but there is also jContractor [9] or iContractor [11] for java and several works have already done for different languages. Furthermore, some studies try to apply design by contract on agent systems like the one done by Christophe GARION [5]. However, whatever the language used, the assertions are expressed as specification code that is a compiled along with the actual implementation code and may disturb it. Furthermore, temporal aspect of the system can't be taken into account with the actual assertions. Finally, as the assertions are expressed on entities codes, the validation focuses on entities' problems and doesn't deals with set of entities or system behaviors.

2.5 Synthesis

All of these researches are interesting and try to validate systems by different ways. However, there are some limitations. Ones may difficult to do due to the complexity of the systems, the others need an important task from users or a lot of informations is needed. Our research takes inspiration from these works and tries to overcome these limitations. The main idea is not to specify all the system but only the critical part of it that we call « properties ». Then, using a multiagent architecture we try to check at run time the information specified. To design this architecture we think it is important to respect a set of rules called WIDE :

- **Weak** : the architecture needs to be weak to make its portability more easier.
- **Independent** : the architecture must be able to validate MAS design on several framework like MADKIT, JADE, or Pœnix also.
- **Dynamic** : the verification needs to be performed dynamically, that is to say at run time.
- **Efficient** : Validating the whole system is not useful, only the specified properties of the system must be be verified.

Considering that a correct system is a system verifying the properties specified by the designer, we will describe in the next section what we call property.

3 Properties

Studying properties is the basis of our researches, so we need to define clearly the property notion. First we will give a general definition for property and secondly we will describe it in a more formal way.

3.1 Definition

Thanks to our study on multiagent system and notably the definition given by Jacques FERBER [3], we define a property as below :

In a multiagent system, we call properties the phenomena, or more generally, the observable effects produced by the agents themselves, by the agents' interactions between each other, and the interactions of the agents with the environment.

To summarize, we can say that:

In a multiagent system, we call property a phenomenon P which:

- is observable;
- is produced by a single agent or by a set of interacting agents;
- brings some elements on the agent's state, on the agent's group or on the environment;
- brings, possibly, some elements on the interactions;
- may predict the aim of an action or of several actions.

Of course, this is a general definition and it will be necessary to fit it with every system. However, we can separate here two kinds of properties: the invariant (or static) properties and the dynamic ones (also called liveness properties). The static ones are properties that are true during the whole execution of the system. They don't change their state. To the contrary, the state of the dynamic properties will change while the system evolve. However, it doesn't mean that the aims of the dynamic properties will change, it just means that they depend of the system's dynamics. Thus, against the static properties, the dynamic ones aren't true or false forever. Their state may change because of interactions. In other words, this is the agents' actions that make dynamic property became true or false.

3.2 Formalism

With the help of the Temporal Logic of Actions defined by Leslie LAMPORT [12], we have defined an early stage formalism for ours properties. As we have seen, there is two types of properties. The temporal definition of static properties **SP** is very simple :

– $\Box P$: the property remains true during all the execution of the program, from the beginning to the end.

For dynamics properties, we define two kinds of properties :

– Delayed Static Property (**DSP**) : $Q \leadsto \Box P$, when the property becomes true, it remains true to the end of the execution.
– Shortlived Dynamic Property (**SDP**) : $Q \leadsto (P \ until \ R)$, the property becomes true and remains true during a finite time. This time may be a time between two events. It is a subtype of the *leads-to property* described by Chandy and Misra [2]

The **DSP** and **SDP** properties follow the same rule :

– $C \leadsto P$, when C becomes true, P becomes true one day. In the sequel, C is called *precondition*.

Noteworthy : Ones may says that a **SP** is a **DSP** with the beginning of the execution as pre-condition. But for a **DSP**, when the precondition becomes true, the property **will** become true **one day**. In other words, the precondition for the **DSP** is a necessary condition but not sufficient to make the property to appear. On the contrary, the beginning of the execution is a condition necessary and sufficient to make the **SP** appear.

Furthermore, in our study, we have seen at least four types of pre-conditions which may be concurrent or exclusive:

– **attribute**, a condition on the state of the attribute;
– **action**, a condition on the fact that an action needs to be activated or not;
– **message**, have we received a message;
– **agent state**, miscellaneous state that doesn't match with the three first elements.

The table below, « Known dynamic property pre-conditions », shows the different values that the known conditions can take and their meanings:

Table 1. Known dynamic property pre-conditions

Types	Values	Meanings
Attributes	true/false =><! value	Boolean attribute comparison values
Action	a+ a- $\emptyset$	active non active none active
Message	r+ r-	received non received

3.3 Properties Extraction

The extraction of the properties is not yet automatized. The designer must define, in the design stage, all the properties to verify. In a previous paper [13], we have defined a methodology, that relies on the work done by Michael WOOLDRIDGE and Nicolas

JENNINGS on the GAIA methodology [22], to extract properties. We propose a two stages study. The first stage, "properties 1", will study properties in the analysis part of the GAIA method. The second one, "properties 2", will give us new properties thanks to the design part and also thanks to "properties 1". The figure below represents the different states of the MAS study.

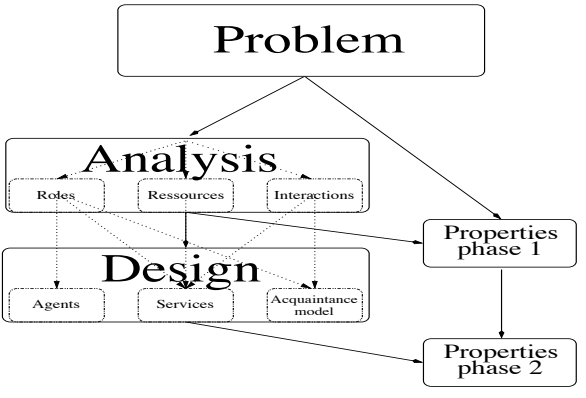


Fig. 1. The properties study

As we can see, the properties are extracted during the analysis and design stages. In fact, it will be at the end of each step of analysis and design that we will define new properties. So, after each step of the MAS construction, we will try to find and complete the set of properties. Therefore, integration of the properties analysis is an additional step for all levels of the MAS construction process.

In section 3.1, we have made a difference between **static** properties and **dynamic** ones. So data needed to describe a property depends on its type. During the extraction process, it is necessary to give at least four data to describe a **static** property:

- **name**: name of the property;
- **type**: static or dynamic;
- **range**: agent, agent group, situated object or environment;
- **action**: actions needed for the validation.

Lets take an example to illustrate **static** property notion. Lets think that an agent in a system has a bag. Clearly the number of elements in this bag can't exceed the capacity of the bag and can't be under zero. Table 2 represent this property where $|R_i|$ represents the capacity of the bag.

To describe a **dynamic** property, we need two more fields:

- **pre-condition(s)**: condition(s) that make dynamic property appear;
- **time**: maximum delay between condition appear and property becomes true.

The notion of time is difficult to clearly define and may change from one system to another. In our theory, section 3.1, a dynamic property becomes true one day once its

Table 2. Example of property

Name: Bag integrity
Type: Static **Range**: agent1
Description: Describes the bag integrity by restricting the number of contained elements.
action: $0 \leq nbElement \leq \lvert R_i \rvert$

precondition becomes true. But because of pragmatic reasons, we need to restrict time between these two events. We have chosen to limit the number of time units and this number is defined by the designer. As time unit may change from one system to another, we don't fix unit value for the time field, it is just a number.

Table 3 represents an example of a dynamic property. An agent has a reserve. When reserve level is equal to zero, the agent needs to refill it and then ask for feel by sending a message.

Table 3. Ask for feel property

Name: AskForFeel
Type: Dynamic **Range**: agent1
Description: ensure that if the level of our reserve is zero, we send a message to ask for feel.
pre-condition(s): reserveLevel = 0 **time**: 5 **action**: send message "empty reserve"

The description field is optional but it may very useful to make your mind clear as we can have very important number of properties. To the contrary, the others fields are critical as we will seen in the next section.

3.4 Specification Languages

Properties are needed to validate multiagent system. Human need to understand and extract properties but some virtual agents need also to understand and use these properties to validate system. So we need to have two specifications languages, one must be understandable by a human (« hight level language »), while the other one must be understandable by an agent at least (« low level language »). The second one is inspired from works on design by contract and notably by those made in java by Murat KARAORMAN *et al.* [8]. We will discuss more about this language later in the paper. Hight level language must be understandable only by human and it might look like what is shown in the table 2. But we want to have more formal representation and we work on a language inspired from the Temporal Logic of Actions (TLA) [12]. The main idea

is to translate hight level language into low level language in an automatic way. For now this work is in embryo but we thing it is a very interesting and useful research way.

Low level language. As we have said, this language is designed in java. Each property is translated into a java method witch returns a string value. The string value must be one of these:

- **"valid"**, if property has been checked and is valid;
- **"non valid"**, if property has been checked and is not valid;
- **"non decidable"**, if one of property's pre-condition is not valid, then the property can't be checked, so it is not decidable.

The method must have zero argument and the name of the property. Let's take the example (table 2) seen in the previous section. This property is represented in our language by this method included in a class named « PropertiesAgent1 »:

```
public String bagIntegrity () {
    if(nbElement >= 0 &&
       nbElement <= max)
          return PState.valid();
    return PState.nonValid();
}
```

As we can see, the name of the method is the same as the property's one, the **range** is directly integrating in the class name and the action is performed by the method's body. In that case, the body may be translated as: if number of bag's elements is between zero and the max value then method return *valid*. Otherwise, the method return *non valid*. Only the **type** field doesn't appear explicitly in the language. But checker agent doesn't really need this information, only the designer needs it to design properties. The fact that one property is dynamic give to the designer that the property depends on time. So it is the designer himself who needs to design property's method according to all property's fields including time. Let's take an example of dynamic property to illustrate these words. Typically the *ask for feel* property seen in section 3.3, table 3, is dynamic. In the java code below, that represent askForFeelProperty, we can see the time notion (arrows →):

```
public String askForFeelProperty () {
  if(reserveLevel == 0) {
     int i = 0;
-->  while(i < 5) {
       if(findMessage("reserve empty"))
           return PState.valid();
-->  pause(1000);
     i++;
     }
     return PState.nonValid();
  }
  return PState.nonDecidable();
}
```

while() and the method **pause** represent the temporal mechanism that we can translate as: if after 5 seconds the property isn't valid, then it is not valid. We can see that the dynamic notion is directly integrated in the body of the method. Furthermore, if the pre-condition (reserveLevel equal zero) is not valid, the property is not decidable.

To conclude, we can say that if designers respect some rules like method's template and return value, we can design every properties as complex as they are. Moreover, as the low level language is in java language, it is well understandable by designer and let the user free to design exotic properties.

In the next section we will describe the architecture of the framework we developed to validate a multiagent system with respect to its properties.

4 PVA : Property Validation Architecture

4.1 Introduction

As we have seen in the section 2.4, the architecture must respect some important rules. So architecture is split into three distinct parts as it is shown in figure 2.

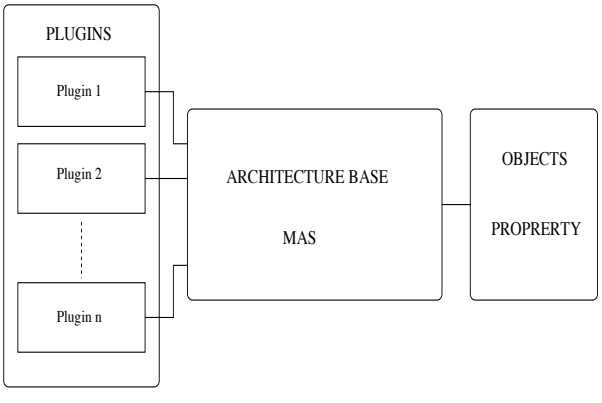


Fig. 2. PVA Architecture

The basis of the architecture is a multiagent system where the checker agents evolve. The checker agents are the entities responsible for the validation of the system studied. Property objects are the essential being for the validation so it needs a particular attention. Finally plug-ins are the pipes between our framework and the system to check or design architecture. All the three part is what we call PVA for Property Validation Architecture.

The architecture basis (section 4.2) allow to have a dynamic and independent framework. The property objects (section 4.3) allow to have the weakness characteristics of the WIDE concept described previously because slight informations is necessary for their efficiency. Finally plug-ins (section 4.4) ensure the efficiency and the portability of the architecture.

4.2 Architecture Basis

The basis of our architecture is a multiagent system. Indeed, problems to study are often complex and using a set of checker agents seems a good choice for at least two reasons. At first, the complexity of the problems is directly fitted into multiagent's complexity.

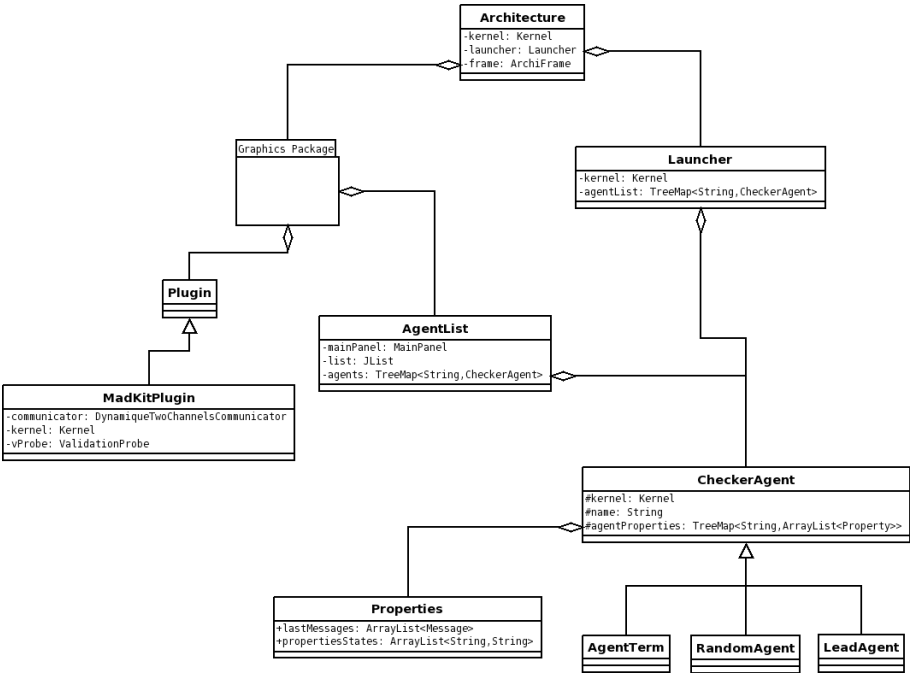


Fig. 3. Architecture UML diagram

Secondly, the efficiency increases because of the number of checking tasks being performed at the same time. For now we use a framework designed by Olivier GUKNECHT and Jaques FERBER [7] as the basis of the multiagent system. Thus we used the microkernel of MADKIT to simulate our checker agents. This seems not fit with the independent rules that we want for the architecture. We do this choice to save time and the fact is that we use only the kernel of MADKIT.

The UML diagram below represent these architecture. *Architecture, Launcher* and *checkerAgent* Classes, their descendants and the whole graphics classes are the architecture basis.

As we can see, we design three types of agents which are all inheriting from CheckerAgent: *RandomAgent, LeadAgent* and *AgentTerm*. All the three have the same goal, that is to say verifying properties of MAS, but their behaviors are different. A checker agent will plug itself to an agent of the system studied and check its properties. The key idea is that a checker agent reads information by looking directly at agent's data to not

disturb its execution. The difference between the three checkers agents is in the way to choose the agent and the properties to check. We define agents' behaviors as follow:

- **RandomAgent**: it evolves in the MAS and chooses an agent in a random way. When it is plugged on an agent to check, it will choose several properties to validate. This agent performs a kind of system audit.
- **LeadAgent**: as a RandomAgent, this one evolves in the MAS to find an agent to check. When it is plugged, it will be lead by the agent in the choice of the properties to check.
- **AgentTerm**: This one is lead by the user. it gives to the user a way to choose the agent of the system to validate and all its properties. The user can choose an agent which seems not working correctly. In other words, this type of agent allows a human assisted watch.

With the agents and the system, we have designed a graphical user interface as we can see in the figure 4.

User interface is divided in four parts. The first one, made of six buttons in the upper left, allows to launch the agents necessary to check the system. Below, we can see the list of launched agents. Clicking to an agent of the list allows to get information on the

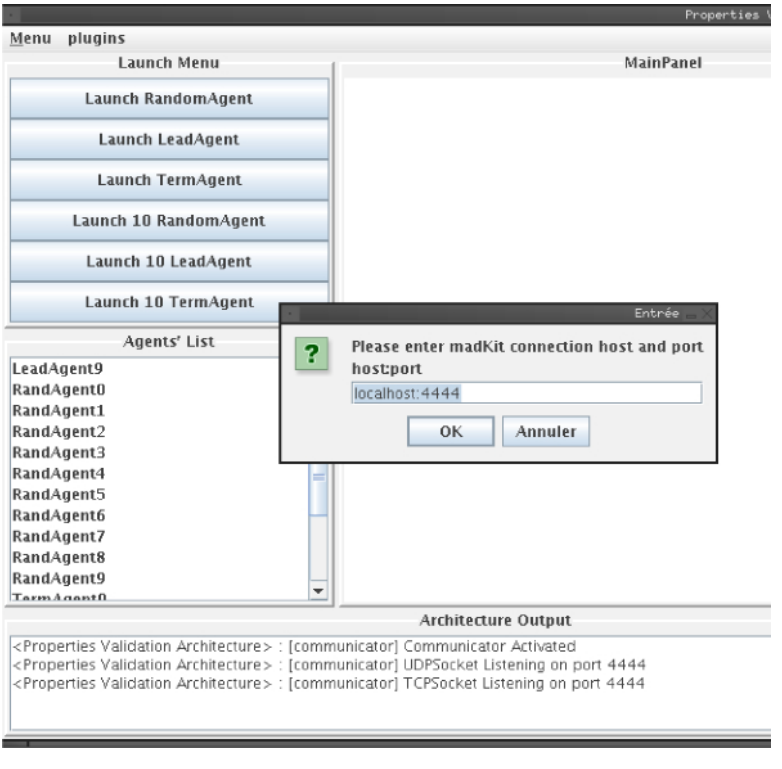


Fig. 4. Graphical user interface

agent checked at this time. This information appears on the *main panel* to the right. The part in the bottom is the output of the system. Finally the pop-up allows to connect with the framework, MADKIT in that case.

4.3 Properties

Checker agents use properties to check systems so we design objects with the informations necessary for validation. In fact, we need to have two generic classes. The first one, the « VerifiableAgent » class , has been designed to make verifiable any agent of the system (including our framework). This class contains validation basic functions and all agents that we want to check need to extend this class. The second one, « Properties », implements some basic functions useful to checker agent. As in design by contract [8], each agent to check is linked to a class that contains all its properties. The class is named with a regular name built as: ***Properties*** key word followed by the agent name (*i.e PropertiesAgentName*). With our previous example, table 2, section 3.3, the class containing all the properties of agent1 is named "PropertiesAgent1". As we have

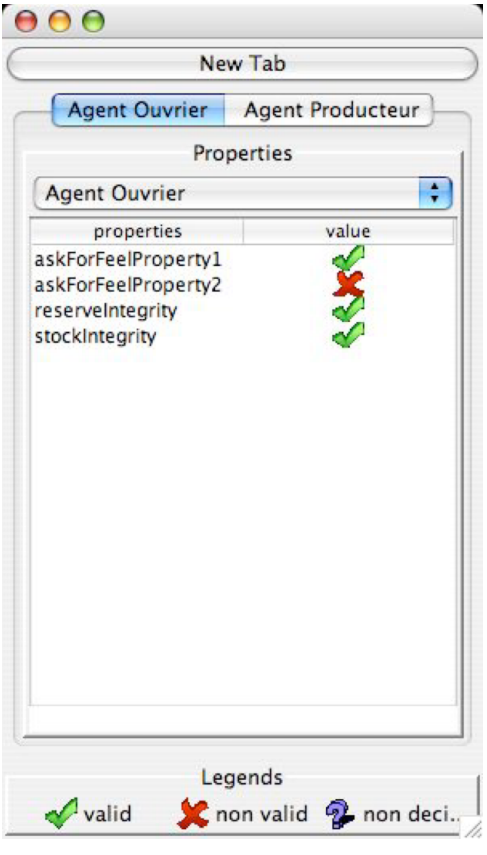


Fig. 5. Agent Random

seen in section 3.2, some earlier study is needed to get properties' information. Once these characteristics are known, we need to complete PropertiesAgent1 class.

To validate our theory and before including it in our architecture, we have designed a prototype using the MADKIT framework. A checker agent is a graphic frame where we can see the agents to check and all their properties as shown in figure 5.

All properties for a given agent and their states are in the main frame. A green check mark (V) means that the property is valid, a red cross (X) that it is not valid and a blue question mark (?) that it is not decidable. In the figure 5 we can see four properties, three of them are valid and one is not. In the top of the main frame there is a scroll menu that contains all verifiable agents. The main frame shows properties of the agent spotted in scroll menu.

4.4 Plug-ins

The long-term goal of our research is to make the architecture independent from any framework. However, the main idea is also to validate MAS design on frameworks such as MADKIT, Phœnix or JADE. So, we need to find a way to check systems designed in such frameworks. Thus, we have chosen to have kinds of pipes between our architecture and frameworks. Such pipes (or plug-ins) need to allow two things at least. First, they must allow communication between PVA and target system. Secondly, it will be necessary to have some system's data from the target system (like value of an attribute), plug-ins must make it possible.

The first plug-in, and the only one developed at the moment, allows the connection with MADKIT.

MADKIT plug-in. MADKIT is a distributed multiagent framework to design multiagent systems. So the communication between several MADKIT is already performed by the framework. The agent *communicator* allows a MADKIT kernel to communicate with another one. As our architecture basis uses a MADKIT kernel, it would be easy for us to communicate with MADKIT. But notice that our long-term goal is to allow the usage of any kernel for the architecture basis. So we can't use this MADKIT kernel to make connection between PVA and MADKIT. So, we use another one MADKIT kernel without any relation with our Architecture basis. Thanks to communicator agent, PVA can communicate with target system designed on MADKIT. We now have the first requirement for the plug-in. The second requirement must allow to have informations about the target system and particularly about the agents of the target system. So we use a set of agents (probe agents) running in the plug-in's kernel will give data (value of properties attributes or any else attributes) needed by checker agents to check properties. Their goal is to read the value of the attribute when they are asked for. They read directly the value in the system and the designer doesn't need to modify any part of its agents' code. Our checker agents, when it's needed, ask for attribute value to them. In fact, the communicator agent is the pipe between MADKIT kernels and probe agents is the pipe between our checker agents and target agents. To summarise, first, target system and PVA seem running in the same kernel thanks to communicator agent. Secondly,

checker agent may have data needed by asking to probe agents. The *communicator* agent, a set of *probe* agents and the micro-kernel define this first plug-in which allows us to validate MAS design in a MADKIT framework.

5 Conclusion

The goal of the PVA architecture we presented is to validate multiagent systems at runtime independently from the platform chosen for its implementation. Contrary to already existing solutions, we made the choice to validate these systems during their execution : in specific cases, it is a quite efficient way. Indeed, we think that there is no better checking that those which are done on systems in operating conditions especially when interested in complex problems solved by MAS. Furthermore, mathematical proof, when they are possible, ought to take a very long time on complex systems. We think that our architecture gives a new solution to the problem of the validation of MAS. Indeed, to integrate the complexity of the problem directly, the architecture is a multiagent system also. Furthermore, we validate the MAS by checking the whole properties specified by the designer which seems more powerful to us. In addition, the idea to use plugins to connect the pva system to various platforms makes the architecture independent and usable by many developers. However, the development of the whole architecture and in particular of the plug-ins is not finished. Indeed, only the MADKIT plug-in is operational yet. In the near future, we will also define more formally the different properties we are interested in.

References

1. Rafael H. Bordini, Michael Fisher, Carmen Pardavila, and Michael Wooldridge. Model checking agentspeak. In *AAMAS'03*, pages 409–416, Melbourne, Australia, 2003. ACM Press.
2. K. Mani Chandy and Jayadev Misra. *Parallel Program Design : A Foundation*. Addison-Wesley, 1988.
3. Jacques Ferber. *Les SystÃŃmes Multi-Agents, Vers une intelligence collective*, pages 13–29. InterEditions, 1995.
4. David W. Flater. Debugging agent interactions: a case study. In *Selected Areas in Cryptography*, pages 107–114, 2001.
5. Christophe Garion and Leendert van der Torre. Design by contract deontic design language for multiagent systems. In *Je sais pas*, pages 107–114, 2005.
6. Tony Garneau and Sylvain Delisle. Programmation orientÃĹe-agent : ÃĹvaluation comparative d'outils et environnements. In *JFIADSMA'02*, 2002.
7. Olivier Gutknecht and Jacques Ferber. The MADKIT agent platform architecture. In *Agents Workshop on Infrastructure for Multi-Agent Systems*, pages 48–55, 2000.
8. Murat Karaorman and Parker Abercrombie. jcontractor: Introducing design-by-contract to java using reflective bytecode instrumentation. *Formal Methods in System Design*, March 2003.
9. Murat Karaorman, Urs Hölzle, and John Bruno. jContractor: A reflective Java library to support design by contract. *Lecture Notes in Computer Science*, 1616:175–??, 1999.

10. Mark Klein and Chrysanthos Dallarocas. Exception handling in agent systems. In Oren Etzioni, Jörg P. Müller, and Jeffrey M. Bradshaw, editors, *Proceedings of the Third International Conference on Autonomous Agents (Agents'99)*, pages 62–68, Seattle, WA, USA, 1999. ACM Press.
11. Reto Kramer. icontract – the java design by contract tool. In *Technology of Object-Oriented Languages and Systems*, pages 295–307, Los Alamitos, California, USA, 1998. IEEEE.
12. Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.
13. Denis Meron, Bruno Mermet, GaÃŐle Simon, and Sylvain Sauvage. Specifying properties of MAS : Towards on the fly architecture. In *SASEMAS'04*, 2004.
14. Bertrand Meyer. *Object-Oriented Software Construction*, chapter 11. Prentice Hall, second edition, 1997.
15. Bertrand Meyer. Applying design by contract. *Computer*, 25(10):40–51, October 1992.
16. Divine Ndumu, Hyacinth Nwana, Lyndon Lee, and Jaron Collins. Vilualising and debugging distributed multi-agent systems. In *the Third Annual Conference on Autonomous Agents*, pages 326–333, 1999.
17. S. Owre, S. Rajan, J.M. Rushby, N. Shankar, and M.K. Srivas. PVS: combining specification, proof checking, and model checking. In Rajeev Alur and Thomas A. Henzinger, editors, *Computer-Aided Verification, CAV '96*, number 1102 in Lecture Notes in Computer Science, pages 411–414, New Brunswick, NJ, July/August 1996. Springer-Verlag.
18. D. Poutakidis, L. Padgham, and M. Winikoff. Debugging multi-agent systems using design artifacts: The case of interaction protocols. In *AAMAS'02*, 2002.
19. David Poutakidis, Lin Padgham, and Michael Winikoff. An exploration of bugs and debugging in multi-agent systems. In *AAMAS'03*, 2003.
20. Steven Shapiro, Yves LespÃĹrance, and Hector J. Levesque. The cognitive agents specification langage and verification environment for multiagent systems. In *AAMAS'02*, 2002.
21. Mark H. Van Liedekerke and Nikolaos M. Avouris. Debugging multi-agent system. *Information and Software Technology Journal*, 37(2):103–112, February 1995.
22. Michael Wooldridge, Nicolas R. Jennings, and David Kinny. The gaia methodology for agent-oriented analysis and design. In *AAMAS'00*, 2000.

On the Application of Clustering Techniques to Support Debugging Large-Scale Multi-Agent Systems

Juan A. Botía, Juan M. Hernansáez, and Antonio F. Gómez-Skarmeta

Departamento de Ingeniería de la Información y las Comunicaciones
Universidad de Murcia

Abstract. This work analyses the problematic of debugging a multi-agent system. It starts from the fact that MAS are a particular type of distributed systems in which the active entities are autonomous in the sense that behavior and knowledge of the whole system is distributed among agents. It situates the problem by firstly studying the classical approaches for conventional code debugging and also the techniques used in distributed systems in general. From this initial perspective, it tries to situate agent and multi-agent systems debugging. It finally proposes the use of conventional data mining tasks like clustering to, by summarising, help in debugging huge MAS.

1 Introduction

Nowadays, there is almost a total lack of tools to assist in the task of debugging and monitoring agent based distributed information systems in where the typical scenario of execution involves hundreds of thousands agents. In such cases, strong and flexible tools are needed to log and recover all necessary data and to analyze it by giving enoughly abstract views. These views should maintain the appropriate abstraction level because, in scenarios involving such a high number of agents, there is a clear necessity of summarizing to gain sight into what is really happening in the system. This paper proposes the use of data mining over agent messages to support this difficult task. Techniques exposed in this paper are implemented in the ACLAnalyser. This tool is described elsewhere [4] and you can find it at the JADE agents platform web page, in the form of an add-on.

The rest of the article is organized as follows. Section 2 introduces the general problem of debugging pieces of software and delimitates the particular issue of debugging agents. Section 3 is devoted to define the general framework we propose here, i.e. to use data mining on agent communication language messages to assist the developer in debugging a MAS (Multi-Agent System). Finally, section 4 outlines initial conclusions and points out future research.

2 Debugging Software Artefacts

Debugging and testing software artefacts is not easy task [22]. Moreover, programming errors may lead to get an information system down virtually all the

R.H. Bordini et al. (Eds.): ProMAS 2006, LNAI 4411, pp. 217–227, 2007.

time, make services offered by a software company unavailable, make not dessired changes to valuable information and, in the worst case, produce wrong outputs.

In debugging software programs we find two different approaches. The first one consists in explicitly modifying the program being debugged to include checks about, for example, the values of critical variables. After that, the analysis process is done without having to execute the code, i.e. statically. This is the approach we may find, for example, in what is called *model checking* [5]. It consists in, given a code to check, to use a graph which represents the different states in which the code being checked may be found. After this graph is built, we use some search algorithm to explore all the possible states trying to find error states. For an example of a real system which systematically checks code on C and C++, please see [16]. A related approach is *program analysis*. It consists in analyzing the code, without having to execute it in order to detect, for example, deadlocks and data races. It detects problematic code regions in the source code, analyzing them and giving an accurate diagnosis about possible errors that may occur during runtime. Please, see [7] for an example of such debugging programs. The second approach consists in dynamically monitoring the program. This is what is called *dynamic checking* [24]. With a dynamic checker, the target code is modified in some way that it checks itself about invariants, and reports on possible violations of the invariants are used to follow the behavior and perform some diagnosis when necessary.

Considering this ideas, the following questions arise. Do model and dynamic checking have any applicability in the context of MAS programming? To what extent is conventional code, debugged by the above mentioned tools, related to that of a typical MAS? Starting from the considerations made in the last paragraph, we may deduce from the last paragraph that model and dynamic checking, may be applied to debug the source code of single agents, provided that they are coded in a language, let it be denoted with $\mathcal{L}$, and we have a checker for $\mathcal{L}$. In this case, we may detect data races and deadlocks in the internals of an agent. At the end of the day, this would be conventional debugging, i.e. it would be like debugging any other program written in $\mathcal{L}$. Notice that this simple analysis has been done on the basis that $\mathcal{L}$ is a general purpose language and not agent oriented like APRIL and JACK may be. An example of a general purpose language used to code agents is Java, as it is used in agent programming environments like JADE, for example. In this case, any existing debugger may be used to analyze the internals of any single agent. In the case that an agent oriented programming language is used, specific debugging techniques either pertaining to model or dynamic checking should be developed first. They should take into account typical agent programming elements like believes, goals, tasks, roles and so on. One good example of such approach is the Tracer tool [12]. It uses reverse engineering and a particular model ckecking to generate *relational graphs* which relate beliefs, intentions and actions. They are also used to generate explanations on actions.

But, would it be possible to apply model and dynamic checking at the inter-agent level? What are the particularities of MAS programming which makes it

different from any other concurrent programming discipline? For the rest of the article, we will limit the answer to that questions with the consideration of an specific, standard and widely used type of agents, FIPA agents. In principle, FIPA agents may be coded in any programming language. For example, FIPA agents from JADE and ZEUS platforms are coded by using Java and agents running at the APRIL Agent Platform are coded using the April agent programming language. This three agent platforms are FIPA (i.e. any agent of the three platforms interoperate with agents in the other two platforms). The only thing that makes all platforms similar is the ACL (Agent Communication Language) [10] they use. FIPA agents talk to each other using a predefined and standard set of communicative acts [6]. In consequence, general FIPA agents systems debugging has to be considered at this level only.

There are two specific reasons for restricting the discussion to FIPA agents. Firstly, we pretend to define a general approach for MAS debugging. We believe that to be general is to be useful for more people. However, there is an important number of different agent theories, architectures and languages. This is something that makes unaffordable a theory of general debugging with some guarantee of success. But, the approach is yet general if we take as a reference, the most widely used framework: FIPA standards are the most widely accepted references to develop agents and multi-agent systems. In this sense, if we focus our attention to FIPA agents, we keep the approach general in the sense that FIPA is the most widely used framework to develop software agents. Secondly, the FIPA ACL may be considered as a very stable and formally defined ACL as all performatives used in the communicative act library come with complete semantics [10].

3 Mining Agent Messages for Debugging

FIPA messages are compound of an envelope and the message content. The envelope is a set of attribute-value pairs with the necessary information for correct delivery and conversation management, as agents structure their communication through conversations following concrete interaction protocols like, for example, the well known Contract Net protocol [20,9]. When designing such kind of protocols, it is possible for the designer to incur in some errors. This kind of errors and techniques for detecting them find their roots in traditional concurrent processes [3] in which code is written with a number of critical sections and the modeling task is usually done with Petri nets [19]. To give an example of such errors, an agent a_i may fall into a deadlock when it keeps waiting for a resource r to be produced by agent a_j and, at the same time, a_j needs another r' to produce r and it is precissely a_i which should produce r'.

In the context of this paper, we are focusing in debugging systems whose design has been completely defined and the MAS has been coded in a concrete language. These kind of errors we mention in the last paragraph will be produced in the design phase when all the coordination protocols for agents interaction were totally specified. In this phase, it is usual to use either finite state automata

or Petri nets to specify the interaction protocols that agents use to interact [21,1]. Hence, is a responsibility of the designer to produce a correct design or either detect these errors. The work of Poutakidis et al. goes in this direction [17,18]. In Prometheus methodology for agent oriented software engineering, interaction diagrams are used to specify interaction protocols and them diagrams are transformed into petri nets which are used to follow dialogues between agents. Another interesting approach for interaction protocols verification consist in using a declarative language to express what is expected from each message, in terms of semantics [2]. This is then used to validate conversations.

So, what kind of systems are the target of ACL mining? This kind of data mining, as we see it, is useful when there is an important number of agents implied in the MAS, conforming an agent society. Moreover, such a MAS compounds a society of hundreds, thousands or even millions agents and an important number of them show a high communication activity (i.e. the number and size of messages exchanged is high during time). The whole picture of debugging, which shows what is the place reserved to ACL messages mining, appears depicted at figure 1.

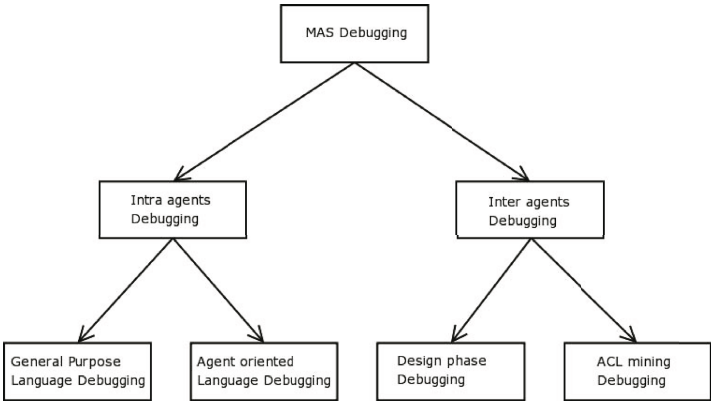


Fig. 1. The different debugging types when considering multi-agent systems

3.1 Source Data

This section will formally define ACL messages mining, $ACLM^2$ for short. We will define what may be the data to be mined in this case and how it will be organized.

Let M be the set of all possible messages that can be exchanged between FIPA agents. An element of M, let it be denoted with m, may be defined as $m = (e, c)$ where the $e = \{v_1, v_2, \ldots, v_n\}$ refers to the envelope of the message by means of the variables representing each of its parameters such as v_i refers to the i-th parameter and contains the value of that parameter in m. Besides, c represents the content of the message. We may also define a session in a run of a MAS, let

it be denoted with S_k for the k-th run, as the total set of messages exchanged in that run. Moreover, S_k may be seen as a data set, compound by tuples of the form (e, c). Tuples in S_k contain categorical data (e.g. the sender and receiver of the message) and numerical data (e.g. the timeout to wait for the next message in the conversation). From now on, we can see each S_k as a relation that could be analysed and it would be possible to extract some knowledge from it, i.e. we can apply conventional data mining [8] techniques to study the activity of the multi-agent system being programmed. Hence, we define $ACLM^2$ as the process of applying conventional data mining techniques to data coming from sessions, with the purpose of locating anomalous behaviors in the execution of the MAS being debugged.

3.2 Data Visualization

One typical data mining task is *complex data visualization* [23]. In this task, data is analysed to find adequate graphical representations which, at a first sight, are capable of representing information in a manner that may allow to obtain quick answers to questions made on source data. We believe that using data visualization in the multi-agent systems development process is useful. And this will be demonstrated through the rest of section 3.

One of these graphical representations is what is called an *agent communication graph*. It consists in a directed graph in which nodes are agents and node i is connected to node j when agent i has sent one or more messages to agent j and j received them correctly. More information may be added to the graph, for example, decorating each arc with the number of messages sent and the total number of bytes transferred. Some of the applications of this very basic view of the whole system are:

- detection of no communication, when expected, between a number of agents,
- detection of excessive amount of bytes exchanged between two or more agents and
- detection of unbalanced execution configurations in which agents from a specific group (or machine) show an amount of activity disproportionate to the rest.

We may find this kind of application view in systems like Zeus agent platform tools. This kind of information representation tools are very convenient when the number of agents in the system being debugged is small (i.e. less than one hundred agents, for example). However, they become useless when this number grows. In this scenarios, when multi-agent systems are really huge, we need more abstract representations. We will now illustrate the situation with a concrete example.

3.3 An Example

This example will consist in using a coordination protocol to distributely decide which agent, among a group of one thousand, will be the leader. This algorithm can be found in [15], pag. 101 and the following is a possible pseudo-code:

```
maxId = ID;
send ID to all acquaintances
on reception of message J do
  if maxId < J then
    maxId = J;
    send J to all acquaintances;
  end
on message from each acquaintance received do
  if maxId = ID then return leader else return follower
```

where ID is an unique identifier which all agents have and which has been randomly set. The leader selection process ends when the agent with the highest ID assumes the leadership, and when this occurs, all agents know which ID is the highest one. An agent knowing that its ID is lower that at least one of the received IDs deduces that it is not the leader. If all IDs received by an agent are lower that its ID, it becomes the leader.

Let n be the number of total agents and m an integer, such as $m << n$. The m represents the number of different disjunct sets in which the n agents are organized. Lets suposse, for simplicity, that the number of agents in each subset is n/m. For each agent set, there will be a group coordinator. This agent has the rest of agents in the same group as acquaintances and these other agents have the coordinator as their unique acquaintance. With this arrangement, the agent communication graph will be a tree, in which the root node is the launcher (i.e. the agent which starts all the others and inform them to start executing) and its direct children are the group coordinators, each one having as children their respective acquaintances.

If we run this example with $m = 500$ and $n = 5$, we may obtain a communication graph like the one appearing in figure 2. This graph was obtained with the ACLAnalyser tool [4]. Quickly, we can conclude that this representation is useless because, in this case *too much information is not information*. However, still a similar graph may be useful in the form of a more abstract representation of the same scenario.

3.4 Clustering Agents for More Abstraction

In this section, we will explain our approach to summarize complex communication graphs, produced in situations where huge MAS runs are represented. The key here is to find which are the most convenient graphical representations or views. Two useful views are the following:

- similar communication activity view: a view in which agents are grouped in the same cluster if they communicate with the same agents. With this view we can group agents showing the same external behavior. Hence, they are similar.
- cooperation activity view: a view in which agents are grouped in the same cluster if they maintain a high communication activity between them. With this other view agents are arranged together because they cooperate.

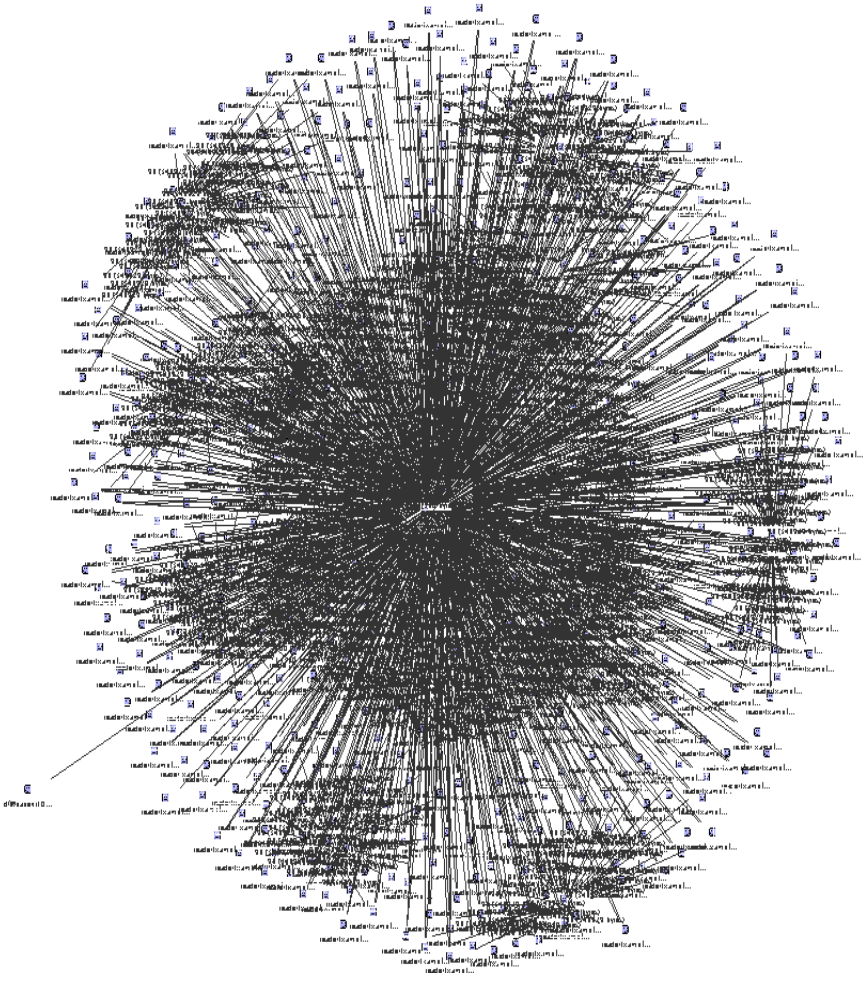


Fig. 2. A *communication graph* compound by all agents in the example

Other useful views would include, for example, the *organizational view* in which, acquaintances are shown related in they maintain some kind of subordinate relation however, this kind of view is out of the scope of the paper.

Obtaining a similar communication activity view. The rest of the section is devoted to explain how to obtain a similar communication activity view. We may perform a clustering over all messages exchanged in such a way that agents were grouped into clusters. Then, a group of k agents belonging to the same cluster would mean that these n agents have been maintaining a similar communication activity (i.e. they have been communicating with the same agents and consequently, they appear grouped).

The particular clustering process we have used here to illustrate the effectivity of $ACLM^2$ is based on the ROCK [11] clustering algorithm. Conventional clustering algorithms detect a priori unknown groupings on data [14]. This grouping is based on a distance measure, typically the euclidean distance. This basic clustering works on continuous data. However, we have categorical data (i.e. messages exchanged between agents in a MAS). This fact brought us to apply ROCK clustering algorithm. This clustering algorithm works with boolean and categorical (i.e. symbols) data. Instead of using a distance, ROCK uses the concept of link. This term is, in turn, based also in the concept of neighbor. Two data points are considered as neighbors if they share some degree of similarity above a certain threshold. This similarity definition depends on each problem. Once neighbors are calculated, two data points have a link between them if they share a common neighbor. After links have been calculated, groupings are compound by data points highly linked (i.e. they all share a high number of neighbors). The number of groups created is a configuration parameter of the algorithm. Hence, the lower the number, the higher the abstraction level we are using to look at the agents society. What would be the definition of neighbor in this case? In this application of ROCK, two agents are neighbors if they share some degree of similarity. Hence, they are neighbors if they share the same links, which means that they separately communicate with the same agents.

The only thing that rests now is to prepare the session to be mined, S_k, by means applying a data transformation in order to generate an appropriate data set for ROCK clustering. This transformation may be defined as the following set

$$T_{rock}(S_k) = \{(a_i, a_j) |\ a_i \text{ sent some } m \in S_k \text{ to } a_j\},$$

where a_i and a_j are agents which participated in S_k. Then we would obtain a S_k^{rock} from $T(S_k)$.

If we perform the ROCK clustering on S_k^{rock}, we will obtain a graph like that appearing in figure 3. This functionality appears also in the ACLAnalyser. Notice that this similar communication activity view represented there is much more informative that the graph of figure 2. The first thing to notice is that the graph has a star topology and that there are six different clusters. Notice that each arc comes with a number showing how many messages are exchanged between agents in the two connected clusters and the size in bytes. If we look inside the cluster labeled with `Group5` (ACLAnalyser allows to click on each cluster to show a list of each agent belonging to it) , we find inside only the leaders of each one of the five groups and also de launcher agent (i.e. the one that executes all agents and then waits for responses on who is the leader from each group coordinator). The rest of the clusters have subordinate agents inside.

Obtaining a cooperation activity view. The cooperation activity view shows how agents in the same cluster maintain a high communication level. In whose case, it is possible for a developer to discover social arrangements of agents showing cooperation clouds.

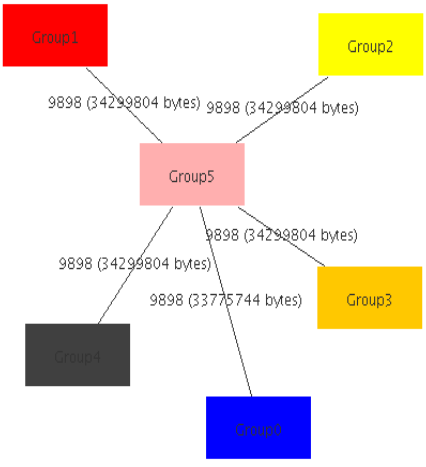


Fig. 3. A *similar communication activity view* in which all agents are grouped into six different clusters

The approach to obtain such a view is similar to the one explained above but the clustering algorithm is not the same. Here, we may use a k-means clustering algorithm [13] which arranges data points into clusters but it locates a centroid in each cluster in such a way that this centroid is the point to what the distance from the rest of the points of the clusters is minium, on average. This clustering algorithm is the most well known grouping technique.

In order to correctly apply the algorithm, we need to find the appropriate data transformation. Given that S_k is the sesion we will mine, we transform it into a S_i^{km} which represents the mentioned cooperation activity view. The idea is that two agents are more near (in the sense of the distance used between data points at clustering) if they exchange more messages. Then, two tuples of S_i^{km} should represent two different agents and no more than one tuple should represent a single agent. Let us suposse that in the S_k run, we have m agents, $\{a_1, a_2, \ldots, a_m\}$. Then, in this case, the transformation of S_k should be defined as

$$T_{km}(S_k) = \{(b_{1,i}, \ldots, b_{m,i}) | 1 \leq i \leq m \text{ and } b_{j,i} \equiv \text{ number of bytes sent from } a_j \text{ to } a_i\}.$$

The kmeans clustering needs a distance metric so we may also define the *communication activity distance*, let it be denoted with $d_{i,j}$ between agents i and j, as

$$d_{i,j} = \frac{1}{1 + t_{a_i}(a_j) + t_{a_j}(a_i)},$$

where $t_{a_i} = (b_{1,i}, \ldots, b_{m,i}) \in T_{km}(S_k)$ and $t_{a_i}(a_j)$ refers to the $j - th$ value of tuple t_{a_i}. Now, it is possible to apply the conventional k-means clustering algorithm.

4 Conclusions and Future Work

In this article we have shown how data mining can be applied to debug highly populated MAS. We have identified the different kinds of debug tasks which may be performed over a MAS, depending on the agent level (i.e. inter and intra agent). We have concluded that data mining may be applied on ACL messages to discover, for example, unknown arrangements of acquaintances in very populated agent societies. We have only used a single data mining task which is clustering. We have shown that, depending on the particular transformations applied to the data obtained in a single run, it is possible to obtain different kinds of groupings of agents in the systems which would help in the debugging process.

Ongoing works include exploring other possible transformations to be applied to S_k to obtain new and useful views. We are also exploring the application of other data well known and widely used mining tasks like classification, regression or association rules mining.

Acknowledgements

Supported by the Spanish Ministry of Education and Science through the Research Project TIN-2005-08501-C03-02 and also by the ENCUENTRO (00511/PI/04) research project of the Seneca Foundation with the CARM.

References

1. S. Abdelwahed and W. M. Wonham. Blocking detection in discrete event systems. In *Proceeding of the American Control Conference*, pages 1673–1678, Denver, Colorado, 2003.
2. Marco Alberti, Marco Gavanelli, Evelina Lamma, Paola Mello, and Paolo Torroni. Specification and verification of agent interaction using social integrity constraints. *Electronic Notes in Theoretical Computer Science*, 85(2), 2004.
3. Gregory R. Andrews. *Concurrent Programming. Principles and Practice.* Addison-Wesley, 1991.
4. Juan A. Botía, Juan M. Hernansáez, and Antonio F. Gómez-Skarmeta. Towards an approach for debugging mas through the analysis of acl messages. *Computer Systems Science and Engineering*, 20, July 2005.
5. E.M. Clarke, O. Grumber, and D. Peled. *Model Checking.* MIT Press, 1999.
6. P. R. Cohen and H. J. Levesque. Communicative actions for artificial agents. In *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95)*, pages 65–72, San Francisco, CA, June 1995.
7. Dawson Engler and Ken Ashcraft. Racerx: Effective, static detection of race conditions and deadlocks. In *Proceedings of the SOSP*, 2003.
8. Usama Fayyad, Gregory Piatetsky-Shapiro, and Padhraic Smyth. Data Mining and Its Applications: A General Overview. In Jiawei Han Evangelos Simoudis and Usama Fayyad, editors, *The Second International Conference on Knowledge Discovery & Data Mining.* AAAI Press, August 1996.
9. FIPA. FIPA Contract Net Interaction Protocol Specification. SC00030, 2002.

10. Foundation for Intelligent Physical Agents. FIPA Communicative Act Library Specification. SC00037, 2002.
11. Sudipto Guha, Rajeev Rastogi, and Kyuseok Shim. ROCK: A robust clustering algorithm for categorical attributes. *Information Systems*, 25(5):345–366, 2000, (citeseer.nj.nec.com/guha00rock.html).
12. D. N. Lam and K. S. Barber. Comprehending agent software. In *AAMAS '05: Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*, pages 586–593, New York, NY, USA, 2005. ACM Press.
13. J. B. MacQueen. Some methods for classification and analysis of multivariate observations. In *5-th Berkeley Symposium on Mathematical Statistics and Probability*, Berkeley, 1967. University of California Press.
14. Tom M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.
15. Jorg P. Muller. *The Design of Intelligent Agents. A Layered Approach*, volume 1117 of *Lecture Notes in Artificial Intelligence*. Springer, 1996.
16. Madanlal Musuvathi, David Y.W. Park, Andy Chou, Dawson R. Engler, and David L. Dill. Cmc: A pragmatic approach to model checking real code. In *Proceedings of the OSDI*, 2002.
17. David Poutakidis, Lin Padgham, and Michael Winikoff. Debugging multi-agent systems using design artifacts: The case of interaction protocols. In *AAMAS'02*, Bologna, Italy, July 2002.
18. David Poutakidis, Lin Padgham, and Michael Winikoff. An exploration of bugs and debugging in multi-agent systems, 2003.
19. Wolfgang Reisig. *Petri Nets, An Introduction*. Springer-Verlag, Berlin, 1985.
20. Reid R. Smith. The contract net protocol: High-level communication and control in a distributed problem solver. In A. H. Bond and L. Gasser, editors, *Readings in Distributed Artificial Intelligence*, pages 357–366. Morgan Kaufmann Publishers, Los Altos, CA, 1988.
21. Agnieszka Wegrzyn, Andrei Karatkevich, and Jacek Bieganowski. Detection of deadlocks and traps in petri nets by means of thelen's prime implicant method. *International Journal of Applied Mathematics and Computer Science*, 14(1):113–121, 2004.
22. James A. Whittaker. What is software testing? and why it is so hard? *IEEE Software*, pages 70–79, January 2000.
23. Graham Wills and Daniel Keim. Data visualization for domain exploration. In *Handbook of Data Mining and Knowledge Discovery*, pages 226–232. Oxford University Press, 2002.
24. Pin Zhou, Fen Qin, Wei Liu, Yuanyuan Zhou, and Josep Torrellas. iwatcher: Simple, general architectural support for software debugging. *IEEE Micro*, pages 50–56, November 2004.

Debugging Agents in Agent Factory

Rem Collier

School of Computer Science and Informatics, University College Dublin, Ireland
rem.collier@ucd.ie

Abstract. The ability to effectively debug agent-oriented applications is vital if agent technologies are to become adopted as a viable alternative for complex systems development. Recent advances in the area have focussed on the provision of support for debugging agent interaction where tools have been provided that allow developers to analyse and debug the messages that are passed between agents.

One potential approach for constructing agent-oriented applications is through the use of agent programming languages. Such languages employ mental notions such as beliefs, goals, commitments, and intentions to facilitate the construction of agent programs that specify the high-level behaviour of the agent. This paper describes how debugging has been supported for one such language, namely the Agent Factory Agent Programming Language (AFAPL).

1 Introduction

The provision of support for debugging MAS is increasingly seen as an important topic of research [10] [17]. Much of the impetus behind this surge in interest is the ever increasing complexity of the problem domains to which agent technologies are being applied. Further, given the slow emergence of prefabricated agent systems that are open and extensible, developers are increasingly required to enhance and adapt these systems to construct new solutions. A side effect that arises from this trend is the need to integration prefabricated agent systems that have been developed using different agent toolkits.

While the issue of how to integrate diverse agent toolkits and architectures is well established through standards bodies such as the Foundation for Intelligent Phyisical Agents (FIPA) [12], the issue of how to debug synthesized multi-agent systems is not. Initial approaches to debugging have, by and large, come from the agent development toolkits research community [14] [19] [20]. While many of these approaches share common ground, for example agent state viewing tools and message monitoring tools, they are all intimately linked to a specific agent toolkit and do not interoperate.

More recently, a number of new debugging tools have begun to emerge [3] [16]. The strength of this new wave of tools is twofold: (1) they have the potential to be independant of any specific agent toolkit or architecture, and (2) they present data at a level of abstraction that scales beyond the equivalent first wave solutions. One such tool is the ACLAnalyser tool, which provides analyses

R.H. Bordini et al. (Eds.): ProMAS 2006, LNAI 4411, pp. 229–248, 2007.

of the interactions that occur during the execution of an agent system. While this system has been implemented and integrated with the JADE framework, its analysis is based on the passing of FIPA-ACL messages. As such, it should be a relatively trivial task to integrate the analyser tool with any FIPA-compliant agent platform, and subsequently provide support for the debugging of agent systems that are implement using multiple FIPA-compliant agent toolkits.

While implementation independence is an obvious strength when charged with the task of analysing large scale agent systems that have been implemented using multiple agent toolkits, it is not sufficient. Implementation independant tools, tend to focus on externally observable behaviours and are able to help developers to hone in on the particular agent or community of agents that are functioning incorrectly. As a result, implementation independant techniques often are not able to provide support for the inspection of the internal workings of those agents. In cases where such support is provided, for example [16], there is a reliance on the developer correctly annotating the internal behaviour of the agent to provide an observable trace of the agents behaviour.

This paper adopts the perspective that the provision of support for debugging of the behaviour of individual agents is best supported through the implementation of debugging tools that are tailored to those agents. Accordingly, this paper focuses on how one such agent development toolkit, known as Agent Factory [5] and its associated agent programming language, the Agent Factory Agent Programming Language (AFAPL), supports the debugging of agent systems. It tackles this issue in two parts: at compile time, and at run time. The former of these perspectives aims to reduce, as far as possible, the number of bugs that appear in the deployed agent system. The latter of these perspectives then focusses upon the support that is provided once the system is deployed.

2 Related Work

The Zeus Toolkit [19] comes packaged with a set of visualisation and debugging tools that provides support at both the agent level and the social level. At the agent level, Zeus includes an *micro tool*, which developers can use to monitor the internal state of the agent. Specifically, it provides a message view (messages sent and received), a summary of actions taken in response to incoming messages, a view of the coordination process for individual goals, a diary of tasks that the agent is committed to, a list of allocated resources, and finally, a summary of the tasks that are being executed or are scheduled for execution [19]. At the social level, the *society tool* provides developers with views of both the social structure of the implemented agent system and the messages that are passed between the agents in that system. Filtering mechanisms allow the developer to focus in on particualr types of interactions within the system.

A more recent tool is *ACL Analyzer* [3]. This tool provides social level support for developers of large multi-agent systems through the provision of visualisation and data mining tools for the analysis of agente interactions. Whilst the tool is implemented for the Jade toolkit [2], it has been designed in a way that it can

easily be plugged into any FIPA compliant agent platform. It works via a logging system that generates a log of all messages that are sent by the agents in the system. A second part of the tool then performs data mining on that log and generates views of the system based on ROCK clustering [13].

A third approach to debugging is the *Tracing Method* [16], a generalised debugging technique that works by capturing dynamic run-time data by logging actual agent data and then using that data to generate models of system behaviour that can be analysed to identify bugs in the agent code. Their approach works by introducing logging statements into the agent code that can be used to generate a model of the agents activities. The advantage of their approach is that it is not tied to any particular toolkit or reasoning model. Instead, the Tracing Method introduces an abstract agent model that logging statements must conform to and then carries out data mining on this abstract model. The model itself includes concepts such as event, action, message, belief, goal, and intention [16]. The main problem with this approach is that, while the approach would work well for systems that are more reactive, there would be some difficulties maintaining that mapping for systems that are more deliberative.

A fourth approach to debugging is the *JACK Design Tracing Tool (DTT)*, is a comprehensive graphical tool that provides support for the visualisation and control of agent behaviours. As with Zeus's micro tool, the DTT provides the developer with a service of views that present aspects of the agents internal state. From the control perspective, the DTT allows developers to start, stop, and step the agent through repeated iterations of its execution algorithm.

There are now a number of agent programming languages, available for download, including 3APL [9], and Jadex [21]. Of these languages, Jadex supports debugging through the use of the Java Logger API, and 3APL comes with a prepackaged interface that provides similar, in a very general sense, functionality to Zeus's micro tool.

The DTT and micro tools are typical of the kinds of tools that are provided by agent development toolkit researchers. Both are similar in function, as both allow the developer to view the internal state of the agent, and both provide some measure of control over that state. A key observation regarding these systems is the level of granularity at which the developer is able to step through the execution of the system. Typically, this granularity is set at the level of one iteration of the underlying execution algorithm. The rationale for this is obvious - one iteration is one full transformation of the agents state. The problem with this, however, is that, when debugging individual agents a developer may need to see how the state evolves during the transformation process.

3 Agent Factory

Agent Factory (AF) is a purpose-built software engineering framework that delivers structured support for the development and deployment of agent-oriented applications [5] [6]. In its earlier form, work on Agent Factory was centered around the objective of delivering end-to-end support for the implementation of agents.

This support included a purpose-built programming language, known as the Agent Factory Agent Programming Language [8] [23], a Foundation for Physical Intelligent Agents (FIPA) standards [12] compliant run-time environment with a prefabricated agent-based application infrastructure (i.e. white and yellow pages services), tool-based support for the fabrication of agents written in that language [22] and a well-defined software engineering methodology that promotes a structured approach to the design, implementation, and deployment of agent-oriented applications [7].

3.1 AFAPL

AFAPL is an agent programming language that supports the development of agents that use a mental state architecture to reason about how best to act. Due to space constraints, only a brief summary of AFAPL is presented here. For an informal overview of the syntax and semantics of the language, the author is directed to [23]. Alternatively, for details of the logic that underpins the syntax and semantics of AFAPL, the reader is directed to [5].

AFAPL supports the fabrication of agents whose mental state is comprised of beliefs, goals, and commitments. Beliefs describe - possibly incorrectly - the state of the environment in which the agent is situated, goals describe future states of the environment that the agent would like to bring about, and commitments describe the activity that the agent is committed to realising. The behaviour of the agent is realised primarily through a purpose-built execution algorithm that is centred about the notion of commitment management [5].

Within AFAPL, commitments are viewed as the mental equivalent of a contract; they define a course of action/activity that the agent has agreed to, when it must realise that activity, to whom the commitment was made, and finally, what conditions, if any, would lead to it not having to fulfil the commitment. Commitment management is then a meta-level process that AFAPL agents employ to manipulate their commitments based upon some underlying strategy known as a commitment management strategy. This strategy specifies a set of sub-strategies that define how an agent adopts new commitments; maintains its existing commitments; refines commitments to plans into additional commitments; realises commitments to primitive actions; and handles failed commitments.

The principal sub-strategy that underpins the behaviour of AFAPL agents is commitment adoption. Commitments are adopted either as a result of a decision to realise some activity, or through the refinement of an existing activity. The former type of commitment is known as a primary commitment and the latter as a secondary commitment. The adoption of a primary commitment occurs as a result of one of two processes: (1) in response to a decision to attempt to achieve a goal using a plan of action, or (2) as a result of the triggering of a commitment rule. Commitment rules define situations (a conjunction of positive and negative belief atoms) in which the agent should adopt a primary commitment.

A key feature of AFAPL, which differentiates it from other agent programming languages, is the inclusion within the language of a set of programming constructs that allow the developer to explicitly specify how each agent can interact with its

```
0  IMPORT com.agentfactory.plugins.core.fipa.agent.Agent;
1
2  ONTOLOGY Ping;
3
4  PLAN ping(?name, ?address) {
5    PRECONDITION BELIEF(true);
6    POSTCONDITION BELIEF(true);
7
8    BODY
9      PAR(inform(agentID(?name, ?address), ping),
10         OR(DO_WHEN(BELIEF(fipaMessage(inform,
11                                       sender(?name, ?addr), pong)),
12                    DELAY(20)),
13            SEQ(DELAY(10),
14                adoptBelief(ALWAYS(BELIEF(unavailable(?name,
15                                                    ?address)))))));
16 }
17
18 BELIEF(monitorAgent(?name, ?address)) &
19 !BELIEF(commitment(?self, ?now, BELIEF(true), ping(?name, ?address))) &
20 !BELIEF(unavailable(?name, ?address)) =>
21 COMMIT(?self, ?now, BELIEF(true), ping(?name, ?address));
22
23 BELIEF(fipaMessage(inform, sender(?name, ?addresses), ping)) =>
24 COMMIT(?self, ?now, BELIEF(true),
25     inform(agentID(?name, ?addresses), pong));
```

Fig. 1. Example Source Code for the Ping Agent

environment. Specifically, AFAPL includes a PERCEPTOR construct, which is used to specify how the agent senses its environment, and a ACTION construct, which is used to specify the primitive actions that each agent can use to effect its environment.

These constructs associate Java classes that implement the sensors and effectors of an agent with the behaviour of that agent which is specified in AFAPL. The set of actuators and perceptors that are specified for a given agent is known as the embodiment configuration of that agent.

By way of illustration, figure 1 presents an example of a simple AFAPL Ping Agent that periodically sends a "ping" message to a specified receiver. It then waits a pre-determined amount of time for a "pong" response from that agent. If no response is received, then the Ping Agent adopts a belief that the specified receiver is unavailable.

The first line of this program (line 0) includes an additional block of agent program code. This code contains, amongst other things, the actions and perceptors needed to support FIPA ACL based agent communication, agent address book management, and the actions and perceptors needed to support locating

and accessing platform services. As is discussed later in section 4, the AFAPL compiler uses this statement to construct a single deployment file that contains all the relevant agent program code.

The second line of the program (line 2) declares an ontology called Ping, which is stored in an ontology file called Ping.ont. This file contains a set of content language terms that must be specified for the ping program. Specifically, this file would include the "monitorAgent(?name, ?address)" term and the "unavailable(?name, ?address)" term.

Lines 4 to 16 specify a plan that is identified by "ping(?name, ?address)". Informally, this plan specifies the following behaviour:

1. Send a "ping" inform message to the specified agent
2. Then wait for one of the following to happen:
   (a) Wait for the receipt of a "pong" inform message from the specified agent, then wait for twenty iterations of the agents interpreter.
   (b) Wait for ten iterations of the agent interpreter and then adopt a belief that the specified agent is now unavailable.

The plan declaration specifies pre- and post- conditions as "BELIEF(true)", which is a default value that causes the conditions to have no effect on the processing of the corresponding commitment. They are used here purely for simplicity. The plan itself, it defined in the BODY part of the plan declaration.

The above example plan introduces a number of AFAPL plan operators. The PAR operation specifies a set of activities that must be carried out, but where the order is not important. The OR operator also defines a set of activities that the agent can carry out in any order. However, in contrast with the PAR operation, the OR operator only requires that one of the activities be completed successfully. The DO_WHEN operator defines an activity that should be carried out only if an associated condition, specified as a belief, is satisfied. Here, the condition is the first parameter, and the activity is the second parameter. Finally, the DELAY operator introduces a delay into the plan. The duration of the delay is specified as an parameter of the operator and is defined in terms of a number of iterations of the agent interpreter.

Finally, lines 18-21 and 23-25 specify two commitment rules that implement the behaviour of of the Ping Agent. The first rule defines when the Ping Agent should send a "ping" inform message. Informally, this rule states that, if the agent believes that it is monitoring an agent, is not currently committed to the "ping" activity (which in this case is the "ping" plan), and does not believe that this agent is unavailable, then it should adopt a commitment to the "ping" activity. The second commitment rule is included to illustrate how an agent that receives a "ping" inform message should respond. In this case, the agent responds by sending a "pong" inform message back to the agent that sent the "ping" message.

3.2 Common Bugs in AFAPL Code

Agent Factory and the associated AFAPL programming language have been employed, over the last 10 years, in the implementation of a nubmer of real-world systems that cover areas as diverse as robotics [11] and mobile computing [18].

As a result of its prolongued use by varous developers, working on these different research projects, it has been possible to identify the three most common categories of bug that developerrs are confronted with:

1. *Incorrectly specified beliefs:* This is perhaps the most common bug that arises when developing AFAPL programs. Simply put, the developer defines a belief that is incorrectly formed. By this, what is meant is that the developer either mistypes the belief, or else specifies a belief that contains an incorrect number of parameters. So, given a correctly formed belief such as "BELIEF(parent(?name))" a developer may: (a) mistype the belief, for example, "BELIEF(Parent(Harry))" - here the capitalised beliefs causes an bug, or (b) may include an incorrect number of parameters, for example, "BELIEF(parent(Harry, George))".
2. *Incorrectly specified activity identifiers:* This second category bug is similar to the first category of bug, with the exception that it applies to incorrectly formed activity (i.e. action and plan) identifiers. In AFAPL, each action and plan is labelled with a unique identifier that is a combination of a name together with a set of arguments (it is something similar to a method signature in Java), for example, "inform(?receiver, ?content)" is the identifier for the inform action, which implements the inform FIPA communicative act. As with beliefs, when specifying a commitment or a plan that uses this action, the developer may either (a) mistype the identifier, or (b) include an incorrect number of parameters.
3. *Mistyped IMPORT and ONTOLOGY statements:* IMPORT and ONTOLOGY statements are used to reference additional files that will ultimately be part of the deployed agent program. In the case of IMPORT statements, these files contain additional AFAPL code that is used by the current agent program. In the case of ONTOLOGY statements, the specified files contain ontologies of terms that can be used when specifying beliefs.

For the majority of cases, these most common AFAPL bugs can be alleviated by performing a series of checks at compile time. The latter errors, however, are more difficult to fix. However, in their worst form, the first two categories of bug can result in the use of the wrong term or activity, or lead to programs that cannot be debugged at run-time. For example, an agent that sends a request instead of an inform, or a belief in which the parameters are specified in the wrong order. As such, this paper describes how support for tracking down and stopping these bugs have been provided through both the AFAPL compiler and the Agent Factory Run-Time Environment.

4 Compile Time Debugging

AFAPL program code is organised into one or more source files, identified by a (.afapl2) extension. This separation of code has been introduced to promote reusability [8]. Support for the reuse of source code is engendered through the IMPORT statement (see section 3.1).

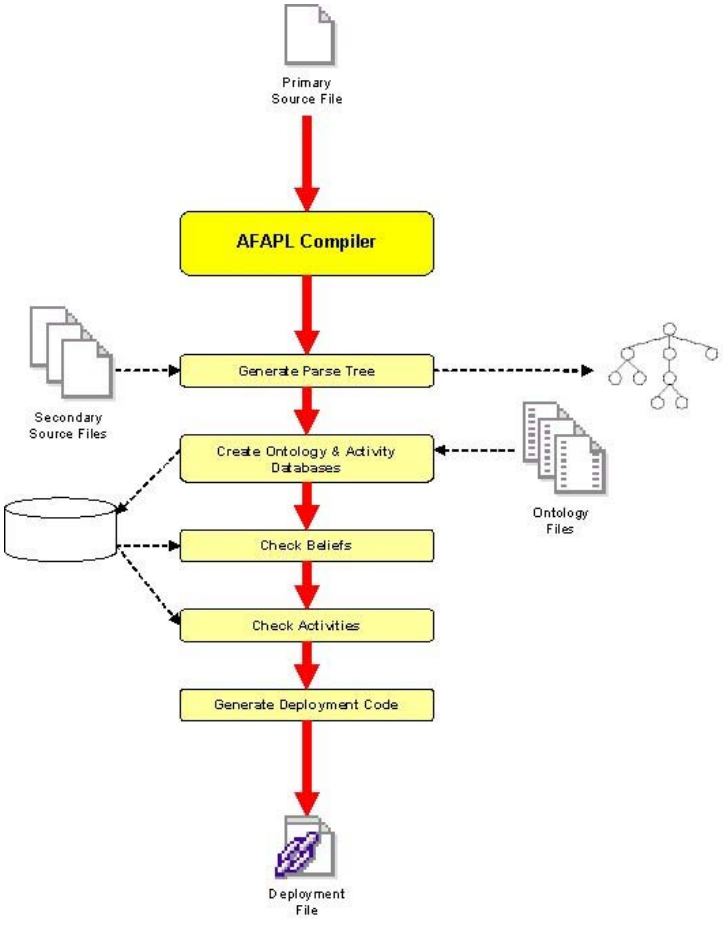


Fig. 2. Schematic of the AFAPL Compilation Process

When a developer wishes to create and deploy an agent, they must select an appropriate source file, which for the purposes of this explaination we will call the primary source file, and compile it into a deployment file (.agent). A schematic of the compilation process is presented in figure 2. At this point in time, the main difference between the source file and the deployment file is that the deployment file is an amalgamation of the various source files that are referenced by the primary source file. This, however, is a temporary solution, and future versions of the compiler will generate a deployment file that is more efficient (in terms of space usage) and more machine-oriented (i.e. can be parsed more effectively than human readable AFAPL code).

In addition to its primary role as the generator of appropriate deployment files, the AFAPL compiler also performs a number of checks that aim to ensure the syntactic and the semantic correctness of the program code. As is common

in compilers for other programming languages [1], the syntactic correctness of AFAPL program code is evaluated through the generation of a parse tree for the primary source file.

Once generated, the initial parse tree is expanded via a tree traversal that seeks out any nodes that contain the IMPORT statement and generates additional parse trees for those nodes. The result of this traversal, known hereafter as a *pass*, is a complete parse tree for the selected primary source file (i.e. a combined parse tree that consists of a parse tree that represents the code from the primary source file together with parse trees that represent the code from each of the secondary source files). If, during the creation of this parse tree, a syntactically incorrect statement is found, the compiler is halted, and a appropriate error message is generated. This error message combines a meaningful error statement together with appropriate contextual information, such as the line number and the surrounding program code.

The successful generation of a complete parse tree implies that the primary and secondary source files are syntactically correct. Once achieved, the compiler moves into a second phase, in which it tries to collect various pieces of information about the program. Specifically, this phase of the compilation process involves two passes that:

- *Pass 1: Generate Ontology Database.* This pass searches the parse tree for nodes that contain ONTOLOGY constructs, and builds a temporary database of content language terms as defined in the specified ontology files.
- *Pass 2: Generate Activity Database.* This pass searches the parse tree for activity (i.e. action and plan) declarations and constructs a temporary database that contains the identifiers of each of the activities found.

If, during the first pass, the compiler locates an ONTOLOGY node that refers to a non-existant ontology file, then the compiler halts and an appropriate error message is generated and displayed. The second pass extracts the identifiers of all activities (actions and plans) that are declared in the parse tree. A requirement of AFAPL is that each activity have a unique identifier. This requirement is enforced during this pass - if the compiler locates an activity identifier that is already in the activity database, then the compiler halts and an appropriate error message is generated and displayed.

Once the compiler has collected this data, it them moves on to a third phase in which it carries out a series of checks that attempt to minimise the potential for semantic errors. Specifically, it performs two additional passes:

- *Pass 3: Check Beliefs.* This pass searches the parse tree for nodes that contain beliefs, and compares them against the terms that are stored within the ontology database.
- *Pass 4: Check Activities.* This pass searches the parse tree for plans and commitments and extracts all activity identifiers from them. It then compares the extracted identifiers against the database of activities that was compiled in pass 2.

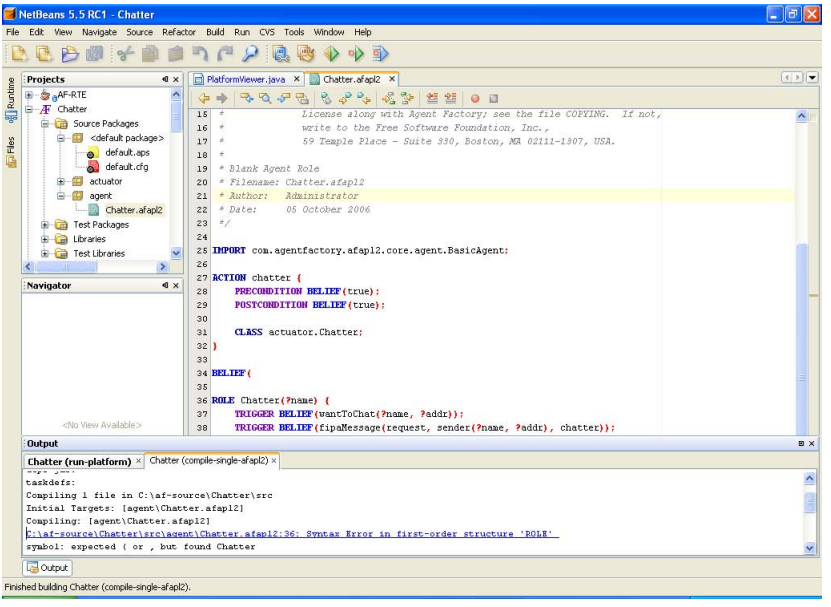


Fig. 3. Screenshot of output of the agent compiler when compiling a program using the AFAPL Netbeans IDE plugin

Passes 3 and 4 are designed to capture, where possible, the first and second categories of bug, as described in section 3.2. In fact, it is for precisely this purpose that these phases have been introduced. Unfortunately, support for the definition of ontologies is a relatively new addition to AFAPL. As a result, we do not enforce bugs identified by passes 3 and 4 as severely as for passes 1 and 2. Rather than halt the compilation process when a bug is detected, the compiler simply generates and displays any appropriate warning message.

The final phase of the compilation process includes just a single pass. This pass is responsible for the generation of a deployment file that is the output of the compilation process.

Figure 3 presents a screenshot of the compiler output when used via the AFAPL Netbeans IDE plugin. As can be seen, AFAPL errors follow the Java error reporting conventions, allowing Netbeans to automatically detect AFAPL errors and provide links to the relevant source code.

5 Run Time Debugging

Compile-time approaches to debugging are effective for catching syntax errors and spotting mistyped or misused terms and activity identifiers. However, they cannot help developers to detect semantic bugs that arise from the use of valid, but incorrect, beliefs and activities. For example, sending a request message instead of an inform message. To detect such bugs, it is necessary to provide

some form of inspection tool that allows the developer to analyse, in detail, the internal workings of an individual AFAPL agent.

Inspection tools are not a new feature of agent toolkits. For example, the Zeus toolkit [19] comes with a pre-packaged micro tool that allows the developer to inspect the state of individual agents and trace their execution. In fact, for many agent toolkits, especially those that can be decomposed into pure Java code, the default inspection and tracing tools are often the ones that are provided with the standard Java Integrated Development Environments. Unfortunately, agent programming languages, such as AFAPL, do not fall into this latter category. As a result, it is essential that custom inspection and tracing tools be provided.

In the context of AFAPL, and Agent Factory in general, support for inspecting and tracing an agent's mental state has long been a key feature [20], which has been realised through the purpose-built *Agent Viewing Tool (AVT)*. This tool provides a graphical interface that presents the developer with a number of views of an agents' internal state. Additionally, the tool provides a set of controls that allow the developer to start, stop, and step through the execution of the agent program. In initial versions of the tool, stepping was fixed at the level of a single iteration of the corresponding interpreter algorithm and views were restricted to simple lists that contained the state of a particular mental attitude (e.g. beliefs, commitments, plans, ...).

While this approach has proved to be a valuable asset in the implementation of agent systems using AFAPL, it is limited in terms of both: the level of detail, which was restricted to a simple list-based views of mental atitudes, and the degree of control, where the developer could only step through complete iterations of the interpreter cycle. Further, while these limitations were not an issue when the complexity of the implemented systems was small (5 to 10 different kinds of agent), they have become significant issues as this complexity has increased.

To cater for this increase in complexity, the AVT has been replaced with a more comprehensive debugging tool, known as the *AFAPL Debugger*. This tool aims to provide a range of views of an agent system, one of which is an agent-level view, which contains an agent inspection tool that is akin to the AVT. Figure 4 presents a screenshot of this new debugging tool with the agent inspection tool open on an agent called "rem".

Access to this revised agent inspection tool is via the "Agent" tab on the left-hand side of the AFAPL Debugger. This tab represents the "agent-level" view of the system and currently contains a list of the agents that make up that system. Other views currently provided include the "Services", which presents usage information for any platform services that are deployed on the agent platform; and the "Histories" view, which provides access to historical executions of the agent platform (i.e. previous runs of the system).

In this section, we describe the various enhancements that have been introduced in our revised agent inspection tool, and which have been developed in an effort to enhance the agent debugging proces:

– To expand the view of the agents internal state to highlight the interplay between various components of that state.

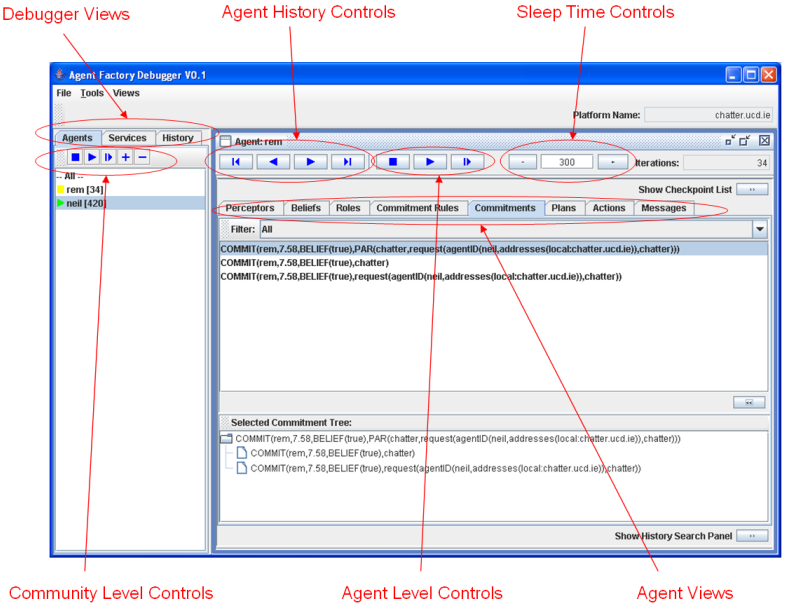


Fig. 4. Screenshot of the AFAPL Debugger with an Agent Inspector open on agent "rem"

- To deliver additional performance information relating to the operation of key components such as the interpreter algorithm, the actuators, and the perceptors.
- To increase the level of control that a developer has over granularity of the step operation.
- To capture the previous states of the agent allowing:
  - support for the analysis of the agents activities over a number iterations of the interpreter (i.e. allowing the developer to step through and view the previous mental states of the agent);
  - history search mechanisms that allow the developer to identify key states of interest (e.g. states in which the agent holds a certain belief, or states in which a certain commitment is adopted).

The first of these improvement relates to practical limitations of traditional agent inspection tools, which present the developer with a limited series of views of the mental state of a given agent. These views often take the form of lists that contain the various atomic sentences that are associated with a given mental attitude after the last iteration. Developers are then expected to infer, via these views of the agnts mental state, how and why each of the sentences associated with a given mental attitude have been adopted/maintained (e.g. in the case of previous versions of AFAPL, the developer must understand how each commitment is adopted through analysis of the beliefs, commitments, and commitment rules of the agent). While this is adequate for small agent programs where there are few commitment rules and as a result it is reasonably easy to

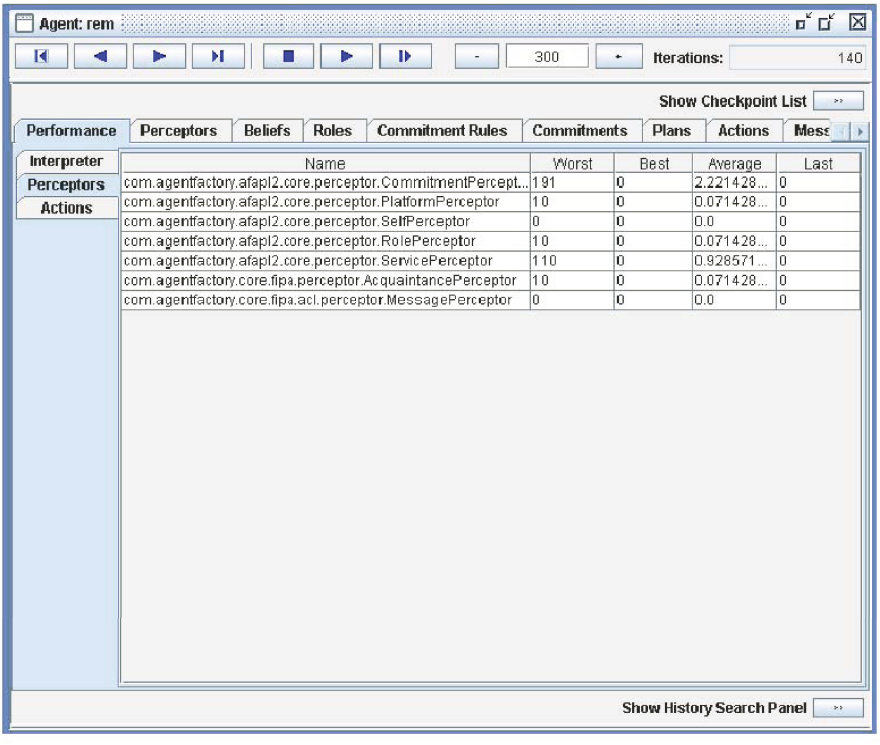


Fig. 5. Screenshot of the AFAPL Debugger with an Agent Inspector open on agent "rem" and the Perceptor Performance view selected

understand why a certain atomic sentence was adopted, it is not the case for larger programs. Accordingly, the AFAPL agent inspector has been upgraded to include additional views that clarify why and how an agent has adopted a given commitment. These new/expanded views include:

- *Perceptor Output View:* A view of the beliefs that are generated by the agents perceptors together with a filter, which can be used to restrict the view to the beliefs that have been generated by a single perceptor.
- *Action Output View:* Displayed debug statements from actions (which, by default, includes the adoption or retraction of any beliefs) and provides a filter that can be used to view debug statements from specific actions.
- *Primary Commitment View:* Lists the current primary commitments of the agent and provides a tree based view that supports visualisation of the commitment tree of a primary commitment (see figure 4).
- *Commitment Rule View:* Provides the standard list based view of the agents commitment rules, but also provides a view of any adopted commitments that were generated by the selected commitment rule.
- *Message View:* Presents a list of all messages sent/received by the agent since the start of the last iteration.

A second set of problems encountered when debugging agent programs using earlier versions of the agent inspector relate to understanding of the performance of the agent. Specifically, many programs reqire the implementation of actuator and/or perceptor units that must carry out some non-trivial task. For example, and agent-based search engine, may require the completion of tasks such as downloading a specified web page, or extracting term frequencies from a downloaded document. When debugging an agent, or attempting to optimise the performance of an agent, it is often useful to know how long such tasks are taking as it directly impacts on the overall performance of the system. To cater for this, a second set of improvements that have been integrated into the latest version of the agent inspector is a number of *performance views* that currently provide the developer with access to information about the operation of:

- the agent interpreter, in the form of timing information for each of the key phases of the agent interpreter algorithm (belief update, role management, and commitment management);
- perceptors, in the form of timing information (currently the worst, best, average, and last performance times) for each perceptor (see figure 5); and
- actions, in the form of timing information (which is the same as for the perceptors) for each action that the agent is able to perform.

As is shown in figure 5, the views that present this information are accessible via the Performance tab of the agent inspector.

The third necessary improvement to the agent inspector has been to increase the level of control that a developer has over the step operation. The step operation, as in Object-Oriented (OO) Programming languages, allows the developer to step through each statement in a program. In many cases, the developer does not wish to check all parts of a given program, but instead wishes to check only a certain part of it. To achieve this, the developer inserts a *breakpoint* just before the part they are interested in. The developer is then able to run the program normally up to the specified breakpoint. Once the breakpoint is reached, the debugger pauses.

As was mentioned in section 2, many of the agent development toolkits provide tool-based support for accessing the agents internal state. Further, some of these environment include a step function that allows the developer to step the agent through successive iterations of its interpreter cycle. While this is good enough for simpler agents, more complex agents are often more difficult to analyse (they have more information, more rules, and the cause of actions can be less obvious). However, to be able to better manage the debugging process, a preferable solution is to provide the developer with more control over how the debugger steps through the interpeter cycle. To achieve this, we introduce the notion of a breakpoint into the AFAPL interpreter.

In sequential and OO programming languages, a breakpoint is a pre-defined point at which a program should pause its execution. Breakpoints are specified in terms of a statement in that program. Typically, once the system reaches a breakpoint, the developer starts to step through the program code, analysing how the state of the program changes.

For AFAPL, we introduce two types of breakpoint:

- *Mandatory Breakpoints* are those breakpoints that are pre-defined for each instance of the AFAPL interpreter and which map onto a number of well defined points at which the agent has reached a partial transformation of its mental state.
- *Optional Breakpoints* are additional developer specified breakpoints that can be set in accordance with the debugging task at hand and which map onto various potential types of event (e.g. the adoption of a specific belief, the triggering of a commitment rule, the performing of an action, ...) within the AFAPL interpreter.

To provide support for these two classes of breakpoint, a generic breakpoint management system has been developed that allows developers to add breakpoints as necessary and to set and unset them as required (we term a breakpoint that has been set to be active, and a breakpoint that has been unset to be inactive). Specifically, a breakpoint manager component has been integrated into the AFAPL interpreter, which is further augmented through the introduction custom Java code that we term *breakpoint evaluators*. A breakpoint evaluator is a snippet of Java code that checks whether any relevant active breakpoints have been reached. It is inserted into the relevant component of the AFAPL Interpreter (e.g. the commitment manager component includes a breakpoint evaluator that is required to check whether or not a given active commitment rule breakpoint has been reached). Whenever a breakpoint evaluator detects that a breakpoint has been reached, the AFAPL interpreter is halted, and the remains halted until the developer either steps or resumes it. In addition to the inclusion of breakpoint evaluators, associated components (i.e. those components of the AFAPL interpreter that implement one or more breakpoint evaluators) must also implement the *Breakpointable* interface. This interface requires the implementaton of a *step()* method which complements the breakpoint evaluator in that it defines how a halted agent is restarted/stepped once the developer wishes to move on to the next breakpoint/continue the execution of the agent. Control of an agent is realised through the three agent level controls (stop, start, and step respectively) that are provided as part of the agent inspection tool (see figure 4).

In its most recent version, the AFAPL interpreter has been updated to include the following types of breakpoint:

- *Commitment Rule Breakpoints*: these breakpoints are reached whenever the corresponding commitment rule is triggered.
- *Belief Breakpoints*: these breakpoints are reached whenever the corresponding belief is adopted.
- *Activity Breakpoints*: these breakpoints are reached immediately prior to the realisation of a commitment to a specified activity.

Visualisation and manipulation of the breakpoints associated with an agent are integrated into the interface of the agent inspection tool. As is shown in figure 6, the agent inspector includes a collapsable view of the breakpoints that

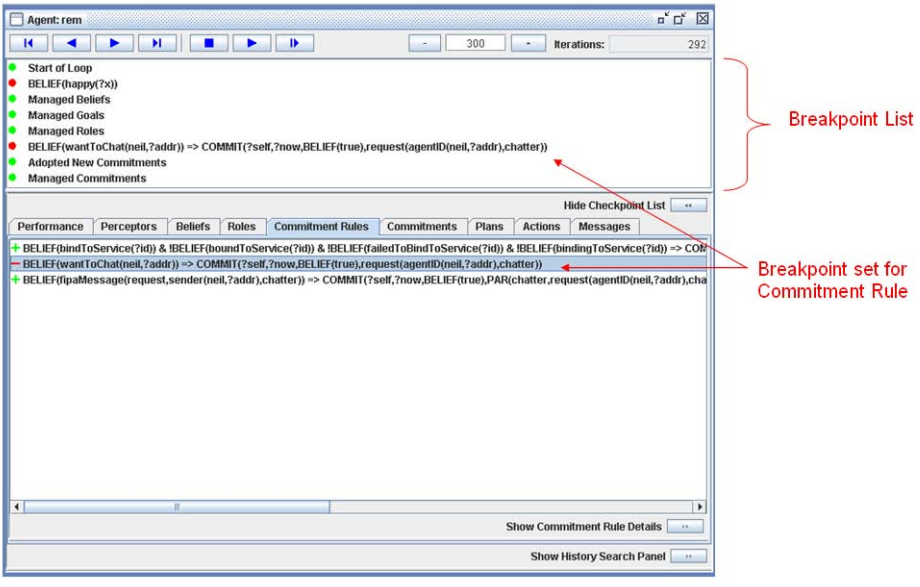


Fig. 6. Screenshot of the Agent Inspector open on agent "rem" showing the breakpoint list together with the Commitment Rule view

are currently set for each agent. An icon is associated with each element in this list that indicates whether or not each breakpoint is set (only set breakpoints are checked). The green icon indicates that the breakpoint is currently set, while the red icon indicates that it is not set.

Individiual breakpoints are added to this list via the associated view. For example, figure 6 also shows the Commitment Rule view of the agent inspector. As can be seen in this view, an icon is associated with each of the three commitment rules. This icon indicates whether or not a breakpoint has been set for that commitment rule. Specifically, a green plus sign indicates that the commitment rule does not have an associated breakpoint while the red minus sign indicates that the commitment rule has an associated breakpoint. To add or remove a commitment rule breakpoint all that the developer needs to do is double click on the associated commitment rule.

The final improvement that has been made to the agent inspector is this introduction of agent histories together with support for history browsing and history searching. Typically, agent inspection tools focus exclusively on the current state of the agent. As such, in order to understand how an agent works, it is necessary to step through each iteration of the agent interpreter. This, by its nature is an intrusive process that results in the agent operating differently to how it would operate if left to run. This results in a scenario where some bugs that are detected during a normal execution of the system cannot be replicated when debugging that system (this is especially true for bugs that arise from concurrency issues).

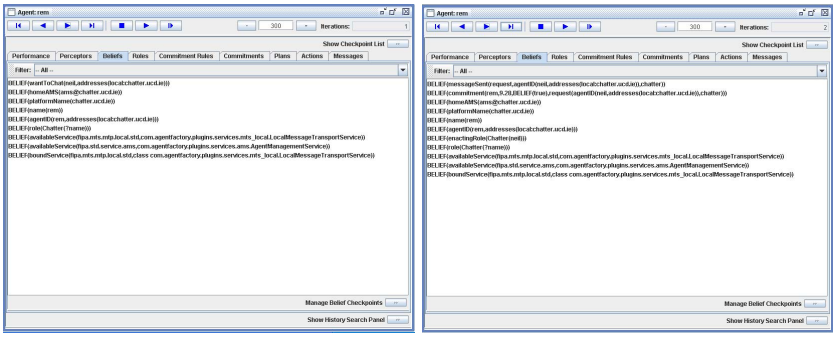


Fig. 7. Screenshot of the Agent Inspector open on agent "rem" after iterations 1 and 2 respectively

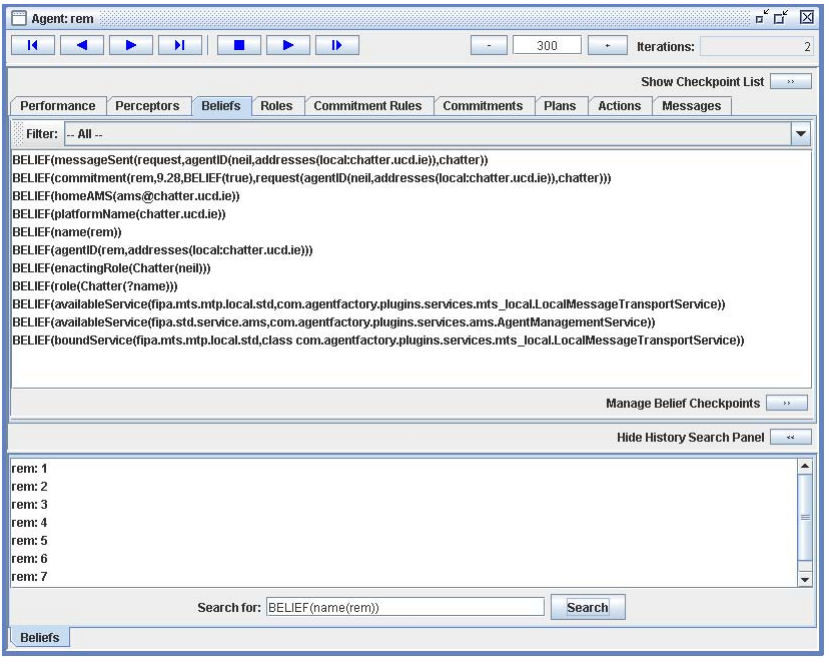


Fig. 8. Screenshot of the Agent Inspector open on agent "rem" showing the results after searching for the belief "BELIEF(name(rem))"

To cater for such issues, a history mechanism has been introduced that records the state of the agent to disk after each iteration of the agent interpreter. A history browsing mechanism is then introduced that allows the developer to browse the historical states of each agent. This browsing mechanism is delivered via a set of historical control buttons that are provided as part of the agent inspection tool (see figure 4). Currently, these control buttons allow the developer

to move to the first iteration, the previous iteration, the next iteration, and the most recent iteration respectively. Selecting one of these options causes the relevant state to be loaded from the disk and presented via the standard agent inspection tool interface.

Finally, to cater for cases where agents have run for some time (i.e. there are a large number of stored states), a search mechanism is also provided. This mechanism is located in a collapsable view at the bottom of the agent inspection tool (see figure 8) and currently provides a mechanism to allow developers to search an agents history for a given belief. Future versions of this tool will extend this search functionality to include other mental attitudes and events.

6 Discussion and Conclusions

The provision of strong support for the development and deployment of agent systems is essential if agent technologies are to gain a wider audience in industry. One area that has, up to now, received little attention is the area of debugging agent systems. Current approaches to debugging can be broadly split into two approaches: community-oriented, and agent-oriented. The former approaches are primarily concerned with monitoring and bug detection in large scale agent systems, while the latter approaches are primarily concerned with the detection of bugs within single agents.

In this paper, we argue that, because of the diversity of implementation approaches, the most rational vision for the future of debugging systems is as a combination of implementation-independant debuggers that provide a more abstract view of agent system and implementation-dependant debuggers that are tailored to a specific agent toolkit. To this end, this paper has described a two prongued approach to debugging agents written in the AFAPL agent programming language. Specifically, we outline how compile time checking of an AFAPL agent program can be used to minimise many types of error that commonly arise when developing AFAPL programs. Following this, we introduce a revised and improved agent inspector tool that provides support for (1) the inspection of the internal state of an AFAPL agent, and (2) monitoring of the performance of the underlying agent components. Additionally, this paper describes a breakpointing mechanism that has been integrated into the AFAPL interpreter. This mechanism can be used to set breakpoints at various pre-determined points in the interpreter cycle. Additionally, the developer can set breakpoints for individual commitment rules, beliefs, and roles. Once reached, a breakpoint causes the interpreter to halt its execution of the agent program. The developer can then use the inspection tool to analyse the state of the agent, and once satisified can step forwards to the next specified breakpoint. Finally, the third enhancement to the agent inspection tool has been the introduction of mechanisms for browsing and searching historical agent states.

While the work presented here focuses on how support for debugging has been provided for a specific agent programming language, we believe that the techniques that have been applied, namely compile-time checking of agent programs

and the introduction of breakpoints as a mechanism for debugging agent programs at run-time, are also applicable to other agent programming languages such as 3APL. Currently, 3APL provides a compiler as part of the 3APL agent platform, however, this compilation step only enforces syntactic correctness of 3APL statements and, to the best of our knowledge, does not check that, for example, referenced Java classes actually exist or that any beliefs and goals specified have the correct number of paramters. Further, as far as the author is aware, while many of this class of agent programming language provides some form of inspection tool, none of them provide performance information or access to previous states of the agent.

References

1. Appel, A., Palsberg, J.: Modern Compiler Implementation in Java (Second Edition). ISBN 0-521-82060-X, Cambridge University Press, UK, 2002
2. Bellifemine, F., Poggi, A., Rimassa, G.: JADE: A FIPA-compliant agent framework. in Proceedings of the 4th International Conference and Exhibition on The Practical Application of Intelligent Agents and Multi-Agents (PAAM), London, UK, 1999.
3. Botia, J.: Debugging hugh multi-agent systems: group and social perspectives. Programming Multi-Agent Systems (PROMAS), Agentlink Technical Forum 3, Budapest, 2005.
4. Busetta, P., Ronnquist, R., Hodgson, A., and Lucas, A.: JACK Intelligent Agents: Components for Intelligent Agents in Java. in AgentLink Newsletter 1, January, 1999.
5. Collier, R.: Agent Factory: An Environment for the Engineering of Agent-Oriented Applications. Ph.D. Thesis, University College Dublin, Ireland, 2001
6. Collier, R., O'Hare, G., Lowen, T., Rooney, C.: Beyond prototyping in the factory of the agents. In Proceedings of the 3rd Central and Eastern European Conference on Multi-Agent Systems (CEEMAS'03), 383-393, 2003.
7. Collier, R., Rooney, C., O'Hare, G.M.P.: A UML-based Software Engineering Methodology for Agent Factory. Proceedings of the 16th International Conference on Software Engineering and Knowledge Engineering (SEKE-2004), Banff, Alberta, Canada, 20-25th June.
8. Collier, R., Ross, R., O'Hare, G.M.P.: A Role-based Approach to Reuse in Agent-Oriented Programming. AAAI Fall Symposium on Roles, an interdisciplinary perspective (Roles 2005), November 3-6, Hyatt Crystal City, Arlington, Virginia, USA, 2005
9. Dastani, M., van Riensdijk, B., Dignum, F., and Meyer, J-J: A programming language for cognitive agents: Goal directed 3apl. Proc. of AAMAS2003, Melbourne, 2003.
10. Dastani, M., Sanz, J.: Programming Multi-Agent Systems (PROMAS) Agentlink Technical Forum Report. in Agentlink News, Issue 19, ISSN 1465-3842, November 2005.
11. Duffy, B.R., O'Hare, G.M.P., O'Donoghue, R.P.S., Rooney, C.F.B., Collier, R: Reality and virtual reality in mobile robotics 1st International Workshop on Managing Interactions in Smart Environments MANSE'99, Dublin December 1999
12. Foundation for Intelligent Physical Agents: The FIPA 2000 Standards. url: http://www.fipa.org

13. Guha, S., Rastogi, R., Shim, K.: ROCK: A Robust Clustering Algorithm for Categorical Attributes. Information Systems, 2000
14. JACK: JACK Intelligent Agents Tracing and Logging Manual. url: http://www.agent-software.com/shared/demosNdocs/JACK_Tracing_Manual_WEB/index.html, 8th June, 2005
15. Jennings, N. R.: On Agent-Based Software Engineering. Artificial Intelligence, 117 (2) 277-296, 2000.
16. Lam, D., Barber, K.: Debugging Agent Behaviour in an Implemented Agent System. In Proc. The Second International Workshop on Programming Multiagent Systems Languages and tools (PROMAS 2004). Held at AAMAS 04, New York, USA, 2004
17. Luck, M., McBurney, P., Shehory, O., Willmott, S.: Agent Technology: Computing as Interaction - A Roadmap for Agent-Based Computing. ISBN 085432-845-9, 2005
18. Muldoon C., OHare G.M.P., Phelan D., Strahan R., Collier R.: ACCESS: An Agent Architecture for Ubiquitous Service Delivery. Proc 7th Intl Workshop on Cooperative Information Agents (CIA2003), Helsinki, 2003.
19. Ndumu, D., Nwana, H., Lee, L., Collis, J.: Visualising and Debugging Distributed Multi-Agent Systems. Proceedings of the third annual conference on Autonomous Agents, Seattle, Washington, USA, pgs 326 - 333, 1999
20. OHare, G.M.P., Collier, R., Conlon, J. and Abbas, S.: Agent Factory: An Environment for Constructing and Visualising Agent Communities. 9th AICS Conference, Irish Artificial Intelligence and Cognitive Science Conference, UCD, Dublin, 19th-21st Aug., 1998.
21. Pokahr A., Braubach L., Lamersdorf W.: Jadex: Implementing a BDI-Infrastructure for JADE Agents. EXP - In Search of Innovation (Special Issue on JADE), Vol 3, Nr. 3, Telecom Italia Lab, Turin, Italy, September 2003, pp. 76-85.
22. Rooney, C F.B., Collier, R.W., O'Hare, G.P.: VIPER: VIsual Protocol EditoR. in 6th International Conference on Coordination Languages and Models (COORDINATION 2004), Pisa, February 24-27, 2004.
23. Ross, R., Collier, R., O'Hare, G.: AF-APL: Bridging princples and practices in agent oriented languages. In Proc. The Second International Workshop on Programming Multiagent Systems Languages and tools (PROMAS 2004). Held at AAMAS 04, New York, USA, 2004

Author Index