State-of-the-Art Survey

LNCS 3390

Ricardo Choren
Alessandro Garcia
Carlos Lucena
Alexander Romano

Software E
Multi-Agen

Research Issues
and Practical Appli

Lecture Notes in Computer Science 3390

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

David Hutchison
Lancaster University, UK
Takeo Kanade
Carnegie Mellon University, Pittsburgh, PA, USA
Josef Kittler
University of Surrey, Guildford, UK
Jon M. Kleinberg
Cornell University, Ithaca, NY, USA
Friedemann Mattern
ETH Zurich, Switzerland
John C. Mitchell
Stanford University, CA, USA
Moni Naor
Weizmann Institute of Science, Rehovot, Israel
Oscar Nierstrasz
University of Bern, Switzerland
C. Pandu Rangan
Indian Institute of Technology, Madras, India
Bernhard Steffen
University of Dortmund, Germany
Madhu Sudan
Massachusetts Institute of Technology, MA, USA
Demetri Terzopoulos
New York University, NY, USA
Doug Tygar
University of California, Berkeley, CA, USA
Moshe Y. Vardi
Rice University, Houston, TX, USA
Gerhard Weikum
Max-Planck Institute of Computer Science, Saarbruecken, Germany

Ricardo Choren Alessandro Garcia
Carlos Lucena Alexander Romanovsky (Eds.)

Software Engineering for Multi-Agent Systems III

Research Issues and Practical Applications

Springer

Volume Editors

Ricardo Choren
Military Institute of Engineering
Systems Engineering Department
Pça General Tibúrcio, 80 - Praia Vermelha, 22290-270 - Rio de Janeiro/RJ - Brazil
E-mail: choren@de9.ime.eb.br

Alessandro Garcia
Carlos Lucena
Pontifical Catholic University of Rio de Janeiro
Computer Science Department
Rua Marquês de São Vicente, 225 - Gávea, 22451-900, Rio de Janeiro/ RJ, Brazil
E-mail: {afgarcia, lucena}@inf.puc-rio.br

Alexander Romanovsky
University of Newcastle upon Tyne, School of Computing Science
Newcastle upon Tyne, NE1 7RU, UK
E-mail: alexander.romanovsky@ncl.ac.uk

Library of Congress Control Number: 2005921208

CR Subject Classification (1998): D.2, I.2.11, C.2.4, D.1.3, H.5.3

ISSN 0302-9743
ISBN 3-540-24843-9 Springer Berlin Heidelberg New York

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

Springer is a part of Springer Science+Business Media

springeronline.com

© Springer-Verlag Berlin Heidelberg 2005
Printed in Germany

Typesetting: Camera-ready by author, data conversion by Olgun Computergrafik
Printed on acid-free paper SPIN: 11392392 06/3142 5 4 3 2 1 0

Foreword

The increasing cooperation and convergence of various kinds of computing entities, i.e., computers, cellphones, personal digital assistants, house appliances, etc., is fundamentally changing the way we view computers and software. The size, increasing complexity and the great potential of future applications for change (e.g., community support, collaborative work and supervision) make centralized and direct control by the programmer nearly impossible. It is thus natural to delegate more autonomy and initiative to various software modules and to provide them with cooperation abilities. Multi-agent systems have been proposed as a conceptual framework to help design and construct such large-scale autonomous and cooperative computing systems.

We can analyze multi-agent systems as a programming paradigm in terms of the evolution of programming in general. We may for instance observe three dimensions of progress: (1) higher levels of abstraction for entities processed or/and exchanged – from bits, and then objects and messages, to agents, intentions and plans; (2) later binding times, i.e., deferring the decision regarding what actual code is to be executed – from procedure call, and then method invocation, to action selection by an autonomous agent; and (3) more flexible coupling between software modules – from procedures, and then objects, components and events, to knowledge-based organizations of agents. Multi-agent systems may thus be seen as an integral part of the evolution of programming.

There is currently a growing body of experience on how to construct multi-agent architectures and platforms based on more conventional technology, objects, Java and components. Interoperability has recently been an important concern, notably through the FIPA Agent Communication Language standard, which builds on object interoperability standards (such as CORBA) and extends their levels of abstraction (e.g., expliciting interaction patterns and protocols, as well as ontologies of knowledge, within a communication). We also see now various attempts and studies for using novel approaches and technologies such as Model Driven Architecture (MDA) and Aspect-Oriented Programming (AOP) in the design and construction of multi-agent systems. Last, the inherent decentralization and autonomy of multi-agent systems also raise new questions about dependability and security.

From the software engineering perspective, we believe the challenges are perhaps even bigger. The decentralized, autonomous and adaptive nature of agents, combined with the large scale, mobility and general dynamicity of their environment support (e.g., ad hoc networks, with associated security and robustness concerns), make it difficult to rely on traditional assumptions of software predictability. In other words, the traditional "defensive/pessimistic" approach – statically safeguarding as much as possible the behavior of the program (through specifications, types, assertions, etc.) – reaches its limits and should be complemented (and not replaced!) by a "proactive/optimistic"

approach, providing the agents with abilities to adapt to unexpected individual and collective behaviors. Another concern is to include also the users, as agents, from the initial stages of the design. Indeed, our ultimate goal is to provide a symbiotic collaboration between artificial agents and human agents, as opposed to confining users, either as supervisers with explicit control or as end-users with little initiative.

For designing practical methodologies, there is still a debate in the community on whether agent-oriented methodologies should be affiliated to current (object-oriented) methodologies or should be deeply restated (for instance by focusing analysis on social concepts such as roles and organization rather than on individual objects or agents). In any case, future methodologies will still need steps (such as analysis, design, modeling, measurements, etc.) as well as related techniques (such as requirements analysis, meta-modeling, notations and metrics). As a consequence, there is currently much activity in studying how such steps or techniques may be partly reused from current technology, adapted or completely rethought.

As a conclusion, in order to achieve these challenges, we need to organize a research community at the crossing of software engineering, programming and multi-agent systems, with a concern for scalability of solutions. This book, the third volume of the very good series on “Software Engineering for Large-Scale Multi-agent Systems,” includes several important studies and proposals along the lines of the various perspectives we just sketched, and thus represents a very good contribution to that research agenda.

Jean-Pierre Briot
Paris, December 2004

Preface

Advances in networking technology in the last few years have turned agent technologies into a promising paradigm for engineering complex distributed software systems. So far they have been applied to a wide range of application domains, including e-commerce, human-computer interfaces, telecommunications, and concurrent engineering. Multi-agent systems (MASs) and their underlying theories provide a more natural support for ensuring important properties, such as autonomy, mobility, environment heterogeneity, organization and openness. Nevertheless, a software agent is an inherently more complex abstraction, posing new challenges for software engineering. Without adequate development techniques and methods, MASs will not be sufficiently dependable, trustworthy and extensible, thus making their wide adoption by the industry more difficult.

Large MASs are complex in many ways. When a set of agents interact over heterogeneous environments, several problems emerge. This makes their coordination and management more difficult and increases the probability of exceptional situations, security holes and unexpected global effects. Moreover, as users and software engineers delegate more autonomy to their MASs and put more trust in their results, new concerns arise in real-life applications. Yet many of the existing agent-oriented solutions are far from ideal; in practice, systems are often built in an ad hoc manner, are error-prone, not scalable, not dynamic, and not generally applicable to large-scale environments. If agent-based applications are to be successful software engineering approaches will be needed to enable effective scalable deployment.

The papers selected for this volume present advances in software engineering approaches to the development of realistic multi-agent systems, demonstrating a broad range of techniques and methods used to cope with the complexity of systems like these and to facilitate the construction of high-quality MASs. Furthermore, the power of agent-based software engineering is illustrated using examples that are representative of real-world applications. These papers describe experiences and techniques associated with large MASs in a variety of problem domains.

A comprehensive selection of case studies and software engineering solutions for MASs applications, this book provides a valuable resource for a vast audience of readers. The main target readers for this book are researchers and practitioners who want to keep up with the progress of software engineering in MASs, individuals keen to understand the interplay between agents and objects in software development, and those interested in experimental results from MAS applications. Software engineers involved with particular aspects of MASs as part of their work may find it interesting to learn about using software engineering approaches in building real systems. A number of chapters in the book discuss the development of MASs from requirements and architecture specifications to implementation.

One key contribution of this volume is the description of the latest approaches to reasoning about complex MASs.

This book brings together a collection of 16 papers addressing a wide range of issues in software engineering for MASs, reflecting the importance of agent properties in today's software systems. The papers presented describe recent developments in specific issues and practical experience. The research issues addressed include (i) integration of agent abstractions with other software engineering abstractions and techniques (such as objects, roles, components, aspects and patterns), (ii) specification and modelling approaches, (iii) innovative approaches for security and robustness, (iv) MAS frameworks, and (v) approaches to ensuring quality attributes for large-scale MASs, such as dependability, scalability, reusability, maintainability and adaptability. At the end of each chapter, the reader will find a list of interesting references for further reading. The book is organized into four parts, which deal with topics related to (i) Agent Methodologies and Processes, (ii) Requirements Engineering and Software Architectures, (iii) Modelling Languages, and (iv) Dependability and Coordination.

This book is a continuation of two previous volumes[1,2]. The main motivation for producing this book was the *3rd International Workshop on Software Engineering for Large-Scale Multi-agent Systems* (SELMAS 2004)[3], organized in association with the 26th International Conference on Software Engineering, held in Edinburgh, UK, in May 2004. SELMAS 2004 was our attempt to bring together software engineering practitioners and researchers to discuss the multifaceted issues arising when MASs are used to engineer complex systems. It was later decided to extend the workshop scope, inviting several of the workshop participants to write chapters for this book based on their original position papers, as well as other leading researchers in the area to prepare additional chapters. Following an extensive reviewing process involving more than 40 reviewers, we selected the papers that appear in this volume.

We are confident that this book will be of considerable use to the software engineering community by providing many original and distinct views on such an important interdisciplinary topic, and by contributing to a better understanding and cross-fertilization among individuals in this research area. It is only natural that the choice of contributors to this book reflects the personal views of the book editors. We believe that, despite the volume of papers and work on software engineering for MASs, there are still many interesting challenges to be explored. The contributions to this book are only the beginning. Our thanks go to all our au-

[1] Garcia, A., Lucena, C., Castro, J., Zambonelli, F., Omicini, A. (eds.): Software Engineering for Large-Scale Multi-agent Systems. Lecture Notes in Computer Science, vol. 2603, Springer, April 2003.

[2] Lucena, C., Garcia, A., Romanovsky, A., Castro, J., Alencar, P. (eds.): Software Engineering for Multi-agent Systems II. Lecture Notes in Computer Science, vol. 2940, Springer, February 2004.

[3] Choren, R. el al.: Software Engineering for Large-Scale Multi-agent Systems – SELMAS 2004 (Workshop Report). ACM Software Engineering Notes, Vol. 29, N°. 5, September 2004.

thors, whose work made this book possible. Many of them also helped during the reviewing process. We would like to express our gratitude to Alfred Hofmann from Springer for recognizing the importance of publishing this book. We also acknowledge the support and cooperation of Anna Kramer and Judith Freudenberger, who helped us in the preparation of this volume. In addition, we would like to thank the members of the Evaluation and Program Committee who were generous with their time and effort when reviewing the submitted papers.

December 2004

Ricardo Choren
Alessandro Garcia
Carlos Lucena
Alexander Romanovsky

Evaluation and Program Committee

P. Alencar (University of Waterloo, Canada)
B. Bauer (Technische Universität München, Germany)
P. Bresciani (Università degli Studi di Trento, Italy)
M. Brian Blake (Georgetown University, USA)
L. Boloni (University of Central Florida, USA)
J.P. Briot (CNRS, France)
J. Castro (UFPE, Brazil)
R. Choren (IME, Brazil)
S. Cost (University of Maryland, USA)
L.M. Cysneiros (York University, Canada)
J. Debenham (University of Technology, Australia)
R. de Lemos (University of Kent, UK)
M. d'Inverno (University of Westminster, UK)
C.A. Fernández (Universidad Politécnica de Madrid, Spain)
M. Fredriksson (Blekinge Tekniska Högskola, Sweden)
A. Garcia (PUC-Rio, Brazil)
P. Giorgini (Università degli Studi di Trento, Italy)
M.P. Gervais (Laboratoire d'Informatique de Paris 6, France)
M.P. Gleizes (IRIT, France)
Z. Guessoum (Laboratoire d'Informatique de Paris 6, France)
B. Henderson-Sellers (University of Technology, Australia)
T. Holvoet (Katholieke Universiteit Leuven, Belgium)
M.N. Huhns (University of South Carolina, USA)
E. Huzita (UEM, Brazil)
C. Jonker (Vrije Universiteit, The Netherlands)
C. Lucena (PUC-Rio, Brazil)
M. Mamei (Università di Modena e Reggio Emilia, Italy)
A. Omicini (Università di Bologna, Italy)
A.D. Pace (UNICEN, Argentina)
A. Rashid (Lancaster University, UK)
A. Romanovsky (University of Newcastle upon Tyne, UK)
G. Rossi (Universidad Nacional de La Plata, Argentina)
C. Rubira (UNICAMP, Brazil)
V.T. Silva (PUC-Rio, Brazil)
A.v. Staa (PUC-Rio, Brazil)
M. Stal (Siemens, Germany)
W. Truszkowski (NASA, USA)
M. Weiss (University of Aberdeen, UK)
A. Zisman (City University, UK)

Table of Contents

Agent Methodologies and Processes

From Object-Oriented to Agent-Oriented Software Engineering Methodologies . . 1
Brian Henderson-Sellers

MASUP: An Agent-Oriented Modeling Process for Information Systems 19
Ricardo Melo Bastos and Marcelo Blois Ribeiro

Composition of a New Process to Meet Agile Needs
Using Method Engineering 36
Massimo Cossentino and Valeria Seidita

A Generative Approach for Multi-agent System Development 52
Uirá Kulesza, Alessandro Garcia, Carlos Lucena, and Paulo Alencar

Requirements Engineering and Software Architectures

A Social-Driven Design of e-Business System 70
Manuel Kolp, T. Tung Do, and Stéphane Faulkner

Systematic Integration Between Requirements and Architecture 85
Lúcia R.D. Bastos and Jaelson F.B. Castro

Integrating Free-Flow Architectures with Role Models Based on Statecharts 104
Danny Weyns, Elke Steegmans, and Tom Holvoet

Aspectizing Multi-agent Systems: From Architecture to Implementation 121
Alessandro Garcia, Uirá Kulesza, and Carlos Lucena

Modeling Languages

CAMLE: A Caste-Centric Agent-Oriented Modelling Language
and Environment 144
Lijun Shan and Hong Zhu

A Formal Approach for the Modelling and Verification of Multiagent Plans
Based on Model Checking and Petri Nets 162
*Hyggo Oliveira de Almeida, Leandro Dias da Silva, Angelo Perkusich,
and Evandro de Barros Costa*

Specification of Role-Based Interactions Components in Multi-agent Systems . . . 180
Nabil Hameurlain and Christophe Sibertin-Blanc

The ANote Modeling Language for Agent-Oriented Specification 198
Ricardo Choren and Carlos Lucena

Dependability and Coordination

A Software Framework for Automated Negotiation 213
Claudio Bartolini, Chris Preist, and Nicholas R. Jennings

Efficient Agent Communication in Multi-agent Systems 236
Myeong-Wuk Jang, Amr Ahmed, and Gul Agha

Adaptive Access Control in Coordination-Based Mobile Agent Systems 254
Christine Julien, Jamie Payton, and Gruia-Catalin Roman

Separation of Concerns for Mechatronic Multi-agent Systems
Through Dynamic Communities 272
Florian Klein and Holger Giese

Author Index 291

From Object-Oriented to Agent-Oriented Software Engineering Methodologies

Brian Henderson-Sellers

University of Technology, Sydney, NSW 2007 Australia
brian@it.uts.edu.au

Abstract. Object-oriented methodologies are well-established and have been used as one input for the creation of methodologies suitable to support the development of agent-oriented software systems. While these agent-oriented (AO) methodologies vary in style and, particularly, in heritage and often with a specific focus (either in terms of domain, application style or lifecycle coverage), for industry adoption it is essential that full lifecycle coverage is achieved in a "standardized" way. One way of achieving some degree of standardization yet maintaining full flexibility is through the use of situational method engineering (SME). With this approach, method fragments are created and stored in a repository. For an individual software development, a subset of these is then selected from the repository and a project-specific (or sometimes organization-specific) methodology is constructed. Here, we demonstrate how this might work by using the OPEN approach that already provides a significant coverage of AO method fragments as well as more traditional OO and pre-OO fragments. Those newer fragments supporting AO approaches are detailed, describing, as they do, emerging substantial support for AO methodological creation from the OPEN repository in an SME context.

1 Introduction

Interest in the creation of appropriate software engineering methodologies for supporting the development of agent-oriented (AO) software systems has shown a rapid increase recently. For many AO methodologists, the object paradigm is seen as a useful precursor. Consequently, many AO methodologies exhibits traits inherited from earlier object-oriented (OO) methodologies – either explicitly or implicitly. On the other hand, some AO methodology writers deny any such influence.

In most cases, the meaning of "AO" in the term "agent-oriented methodology" means a methodology to be used for building agent-oriented software systems. However, in one case (Tropos, e.g. Bresciani *et al.,* 2004), it is used to mean that the agent concept is used in the conceptual underpinning of the methodology itself.

It should be noted that although we use the term "methodology", which means a full description of process, people, social structures, project management, modelling language, products etc. (e.g. Henderson-Sellers, 1995; Rolland and Prakash, 1996), some of the methodologies referred to in this paper provide only partial support – perhaps in terms of only addressing analysis and design (as does Gaia e.g. Wooldridge *et al.*, 2000; Zambonelli *et al.*, 2003) or omitting any discussion of the

R. Choren et al. (Eds.): SELMAS 2004, LNCS 3390, pp. 1–18, 2005.
© Springer-Verlag Berlin Heidelberg 2005

"people element", for instance, MaSE (DeLoach, 1999) or AOR (Wagner, 2004), the latter being primarily a modelling language.

In this paper, we examine the evolution of agent-oriented methodologies and their relationship to earlier AO and OO methodologies leading to suggestions for future AO methodology support that may be of interest to industry. In Section 2, we analyze the various extant AO methodologies in terms of their OO/non-OO lineage. In Section 3 we debate the difference between a "one-size-fits-all" methodological approach versus a more flexible approach, the latter using situational method engineering (SME). The SME approach is then illustrated by a case study (Section 4) using the OPEN metamodel and repository of method fragments (Graham *et al.,* 1997; Henderson-Sellers *et al.,* 1998), recently extended to offer wide support for agents.

2 Methodology Genealogy

The development of AO methodologies has taken many routes. Some methodologists have based their methodological approach on an Artificial Intelligence or Knowledge Representation; others have commenced with basic definitions of objects and then asked what modifications are necessary to support agents; others have commenced with an established OO methodology and asked how agent support can be grafted on.

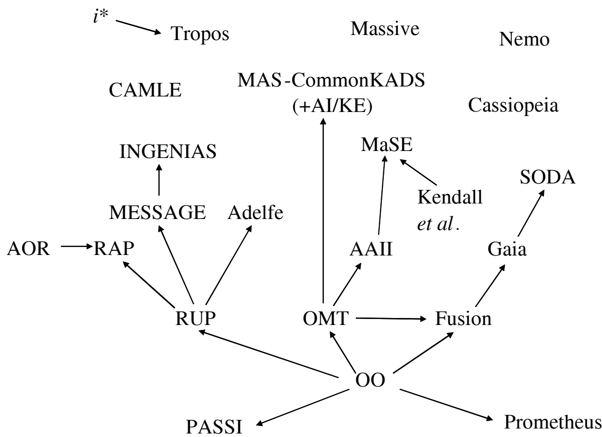


Fig. 1. Genealogy of various AO methodologies and their relationships to OO methodologies

Figure 1 graphically depicts some of these linkages and influences. OO methodologies such as RUP (Kruchten, 1999), OMT (Rumbaugh *et al.*, 1991) and Fusion (Coleman *et al.*, 1994) have all been used by various AO methodology groups as the basis for agent-oriented extensions. RUP has formed the basis for Adelfe (Bernon *et al.*, 2002) and also for MESSAGE (Caire *et al.,* 2001), which, in turn, is the basis for INGENIAS (Pavon *et al.*, 2005) and, more recently, RUP has been a useful input to RAP (Wagner and Taveter, 2005), a direct descendant of AOR (Wagner, 2003). OMT is said to have directly influenced MAS-CommonKADS (Iglesias *et al.*, 1996, 1998), which merges these OO ideas with concepts from AI and Knowledge Engineering, as

well as the AAII approach (Kinny *et al.*, 1996) which, in turn, is said to have been a major influence on MaSE (DeLoach, 1999; Wood and DeLoach, 2000). Fusion has strongly influenced Gaia which, in turn, has influenced SODA (Omicini, 2000). Prometheus (Padgham and Winikoff, 2002a,b) is a fully AO methodology but states that one should use UML-style diagrams when appropriate rather than "reinvent the wheel". All of these AO methodologies are "standalone" – effectively "one size fits all" – approaches.

Other methodologies in Figure 1 do not acknowledge any influence from any OO approach – although clearly some have had an implicit influence. Tropos is said to be based on *i** (Yu, 1995) and has a distinct strength in early requirements modelling. Its use of the *i** modelling language gives it a different look and feel to those that use Agent UML (AUML: Odell *et al.*, 2000) as a notation. It also means that the non-OO mindset permits users of Tropos to take a unique approach to the modelling of agents in the methodological context.

There is no obvious, explicit evidence of an OO influence in the published versions of Nemo (Huget, 2002), MASSIVE (Lind, 2001), Cassiopeia (Collinot *et al.*, 1996; Collinot and Drogoul, 1998), PASSI (Cossentino and Potts, 2002; Burrafato and Cossentino, 2002)[1] and the work of Kendall *et al.* (1996). CAMLE (Shan and Zhu, 2004) does, however, draw some parallels, particularly between a CAMLE caste and an OO class and with respect to UML's composition and aggregation relationships.

Several authors have made direct comparisons of these (and other) AO methodologies. Cernuzzi and Rossi (2002) proposed a framework containing a set of internal attributes (autonomy, reactivity, proactiveness and mental notions), a set of interaction attributes (social ability, interaction with the environment, multiple control, multiple interests and subsystems interaction) and four other requirements (modularity, abstraction, a system view and communication support). They used this framework in a case study to evaluate a BDI focussed methodology (Kinny *et al.*, 1996, variously referred to as AAII or BDIM) and MAS-CommonKADS (Iglesias *et al.*, 1998) both qualitatively and, with an appropriate set of metrics, quantitatively. This study and other comparative evaluations of both AO and OO methodologies were used as input to the framework proposals of Dam and Winikoff (2004) who proposed four categories: concepts, modelling language, process and pragmatics. Their contribution is that the evaluation was not only done by the authors but by surveying a set of students who had used the case study methodologies (MaSE, Prometheus and Tropos) on a design problem of a mobile travel planner. The same four categories were used by Sturm and Shehory (2004) and used to evaluate Gaia (as a single example) using a seven point quantitative metric scale. The framework of Tran *et al.* (2003) also has four categories but these are said to be process-related (15 criteria), technique-related (5), model-related (23) and other supportive features (8). The framework was applied by Tran *et al.* (2004b) to five well-referenced AO methodologies – namely MaSE, Gaia, BDIM, Prometheus and MAS-CommonKADS. Different ordinal scales are used for the several criterion sets. A more extensive set of results (the evaluation of 10 AOSE methodologies) is found in Tran and Low (2005).

[1] A more recent manuscript in preparation does, in fact, acknowledge influences from object technology.

3 Specific or General Methodologies?

To support any software development, there would appear to be (at least) three options: (i) create a suite of inflexible methods, each of which is highly tuned to specific operating conditions; (ii) create a single all-inclusive methodology and then permit some removal of unwanted elements (sometimes known as method tailoring); and (iii) create not a methodology but a methodological framework underpinned by the concepts of situational method engineering (see Section 3.2 below) that permits the construction of multiple, specifically configured methodologies – one for each particular operating situation.

Using a suite of methodologies provides perfect alignment with the problem at any given time but, as situations change, provides no route for migration from the current methodology to a second in the suite, however perfect that second one might be for the new problem space. Thus, there is no possibility of encouraging the valuable process of Software Process Improvement or SPI, as advocated by e.g. CMM or SPICE (ISO 15504) because there is no route between these methodological "islands" (Figure 2).

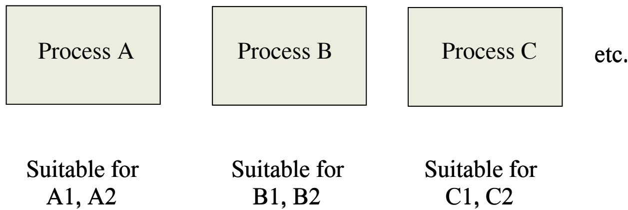


Fig. 2. "Islands" of methodology provide no route to migrate between them and hence there is no potential for SPI

Using a comprehensive methodology typically requires users to understand all elements of the approach before beginning a reduction programme i.e. eliminating the elements of this comprehensive methodology that are not needed for this specific project. This can mean wasted effort and such so-called heavyweight methodologies are often seen as anathema to contemporary problems (Avison and Fitzgerald, 2003) which are often said to require more "agile" approaches to software development.

In many ways, the "best of both these worlds" can be achieved through the third option and that is the one we will explore in this paper in more detail below (Section 3.2) following a brief overview of some of the AO methodology "islands" (Section 3.1) currently available for use.

3.1 Specific AO Methodologies

Many individualistic methodologies have been formulated and published. Here, we review briefly a small selection, focussing on those that have already been analyzed in order to extract method fragments (see Sections 3.2 and 4). Each description below emphasizes the *agent*-oriented aspects of that methodology, needed to go beyond the basic object-oriented concepts that many of them utilize.

Prometheus (Padgham and Winikoff, 2002a,b) is an agent-oriented methodology that reuses as many appropriate elements as possible from object technology including several UML diagram types. In the first phase (of three) of systems specification, the basic functionality of the system is identified, using percepts (inputs), actions (outputs) and any necessary shared data storage. This is followed by the architectural design stage; here, the agents and their interactions are identified. Finally, there is the detailed design phase in which the internal details of each agent are addressed.

MASE (DeLoach, 1999; Wood and DeLoach, 2000) is drawn from the legacy of object-oriented methodologies such as OMT together with influences from the more recent UML as well as pre-existing work in the realm of agents and multiagent systems e.g. Kinny *et al.* (1996) and Kendall and Zhao (1998). It aims to guide the designer through the multiagent-system development process from an initial system specification to a set of formal design documents. It has two phases: analysis and design. The former deals with the specification of system goals, use cases, sequence diagrams, roles and tasks, while the latter uses the analysis phase's outputs to design agent classes, agent interactions and agents' internal components. It is also well supported by a software tool.

Gaia (Wooldridge *et al.,* 2000) views the process of multi-agent system (MAS) development as a process of *organizational design*, where the MAS is modelled as an organized society with agents playing different roles. The methodology allows a developer to move systematically from a statement of requirements to a design that is sufficiently detailed that it can be implemented directly. It supports both macro (societal) and micro (agent) aspects of MAS design, and is also neutral to both application domain and agent architecture. The newest version of Gaia (Zambonelli *et al.*, 2003) extends the original version with various organizational abstractions, enabling it to be used for the design of open MAS (which was not achievable previously).

Cassiopeia (Collinot *et al.*, 1996) provides an (arguably incomplete) methodological framework for the development of collective problem-solving MASs. Cassiopeia assumes that, although the agents can have different aims, the goal of the designer is to make them behave cooperatively. It adopts an *organization-oriented approach* to MAS design, as do some other AO approaches, viewing an MAS as an organization of agents that implement/encapsulate *roles*. These roles not only reflect the agents' individual functionality, but also the structure and dynamics of the organization of the MAS.

MAS-CommonKADS (Iglesias *et al.*, 1998) is an agent-oriented methodology that supports the development of MAS from the conceptualization phase through to a detailed design that can be directly implemented. The methodology integrates techniques from a well-known knowledge-engineering methodology, CommonKADS (Schreiber *et al.*, 1994), with those from OO methodologies (e.g. OMT, OOSE and RDD) and protocol engineering. The main modelling concepts in MAS-CommonKADS are agent, knowledge, organization and coordination.

Agent Factory (Collier *et* al, 2003, 2004) is a four-layer framework for designing, implementing and deploying multi-agent systems. It contains (i) an agent-oriented software engineering methodology, (ii) a development environment, (iii) a FIPA-compliant runtime environment and (iv) an agent programming language (AF-APL); with a stated preference for the BDI agent architecture according to the analysis of (Luck *et al.*, 2004). By employing UML and Agent UML, the Agent Factory method-

ology provides a visual, industry-recognized notation for its models - regarded by its authors as a major advantage over other approaches, such as Gaia (Wooldridge *et al.*, 2000) and Tropos (Bresciani *et al.*, 2004), which have non-standard (i.e. non-UML compliant) notations. These models are capable of promoting design reuse (via the central notion of *role*) and being directly implemented by automated code generation (Collier *et al.*, 2004).

CAMLE (Shan and Zhu, 2004) is described as a caste-centric agent-oriented modelling language and environment. It is caste-centric because *castes*, analogous to classes in object-orientation, are argued to provide the major modelling artefact over the lifecycle by providing a type system for agents. A significant difference is claimed between castes and classes: while objects are commonly thought of as statically classified (i.e. an object is created as a member of a class and that is a property for its whole lifetime), agents in CAMLE can join and leave castes as desired, thus allowing dynamic reclassification. CAMLE provides a graphical notation for caste models (similar to class models in OO methodologies), collaboration models and behaviour models. Caste diagrams also include support for the non-OO relationships of congregation, migration and participation. CAMLE relies heavily on the fact that an information system already exists when a new project is started, so that the new system is designed as a modification to the current one. Although this situation is indeed common, the construction of systems from scratch also happens. CAMLE, however, seems to ignore this possibility.

Tropos (Perini *et al.*, 2001; Castro *et al.,* 2002; Bresciani *et al.,* 2004) was designed to support agent-oriented systems development with a particular emphasis on the early requirements engineering phase. The stated aim was to use agent concepts in the description and definition of the methodology rather than using OO concepts in a minor extension to existing OO approaches. Tropos takes the BDI model (Rao and Georgeff, 1995; Kinny *et al.*, 1996), formulated to describe the *internal* view of a single agent, and applies those concepts to the *external* view in terms of problem modelling as part of requirements engineering. It also relies heavily on the *i** framework of Yu (1995) for concepts and notation.

In summary, there is a tendency to reuse significant portions of object-oriented methodological approaches, supplementing them with a new focus on organizations, social interactions, proactivity and roles. There is still discussion about the extent to which UML can be useful. Several AO methodologies use existing UML or, often, AUML diagrams but, at the same time, find deficiencies for which they supply new diagrammatic representations. In particular, there is still argument as to whether an agent concept could be added to the UML metamodel simply as a subtype of the Classifier metaclass or whether a totally different conceptualization is needed (e.g., Silva and Lucena, 2004).

3.2 General Methodologies – The Use of Situational Method Engineering

In contrast to an individual AO methodology, we now explore the third option of creating a methodological framework. In particular, in this section we outline the concepts of situational method engineering or SME (Kumar and Welke, 1992; Brinkkemper, 1996; Ter Hofstede and Verhoef, 1997).

SME suggests that the elements of a methodological can be modularized and encapsulated as “method fragments” (van Slooten and Hodes, 1996). The method fragments can then be connected to form larger fragments and finally the whole methodology. There is thus no initial or default methodology stored in the method repository or methodbase (e.g. Brinkkemper, 1996; Ralyté and Rolland, 2001) and indeed the methodbase may contain conceptual fragments originating from various sources. Ideally, the method fragments should all be instances of a concept captured in a metamodel underpinning the methodbase (Ralyté and Rolland, 2001; Henderson-Sellers, 2003). The metamodel provides essentially a set of rules and prescriptive descriptions of all the kinds of method elements permissible within the methodbase.

The challenge for the method engineer is to select appropriate and compatible fragments and to construct the final methodology (e.g. Wistrand and Karlsson, 2004). This may be from scratch or as an extension to an existing methodology (Ralyté *et al.*, 2003). Thus, construction guidelines (e.g. Klooster *et al.*, 1997; Brinkkemper *et al.*, 1998; Rolland *et al.,* 1999; Ralyté and Rolland, 2001; Ralyté *et al.*, 2004) are critical in the SME approach. Creating a project-specific methodology is currently one of the more difficult and time-consuming jobs of the method engineering approach, since the method engineer has to understand the methodology, the organization, the environment and the software project in order to select the appropriate fragments from the repository to use on the project as well as understanding the rules of construction. Traditionally, this process is carried out using predefined organizational requirements and the experience and knowledge of the method engineer or process engineer (e.g. Fitzgerald *et al.*, 2003), although significant tool support is likely in the near future (Saeki, 2003; Wistrand and Karlsson, 2004).

4 Case Study: Supporting Agent-Oriented Software Engineering Using the OPEN Framework

One example of a method engineering approach that can encompass both object-oriented and agent-oriented methodological thinking is the OPEN Process Framework or OPF (Firesmith and Henderson-Sellers, 2002). OPEN adopts a framework approach based on an underpinning metamodel, and has recently been extended from its original object-oriented base to include methodological support for agents (see, e.g., Henderson-Sellers and Debenham, 2003). As with any method engineering approach, OPEN aims to provide a repository of method fragments that will offer direct as well as extensible support for the construction of individually tailored methodologies for use in both industry and research environments.

OPEN’s method fragments are generated directly from its metamodel (Figure 3) and stored in the OPF repository. To create a situated methodology, various method fragments are then chosen from this repository and combined to describe the process, associated people and social issues, deliverables and so on – each of which is defined formally by the corresponding metalevel element in the metamodel (Figure 4). In other words, a full-scale and comprehensive methodology can be constructed from the repository fragments. This could have an object-oriented, an agent-oriented or even a traditional (procedural-focussed) bias.

Using the tenets of SME outlined above, such a methodology can be specifically constructed and tailored towards a specific project or a specific organizational

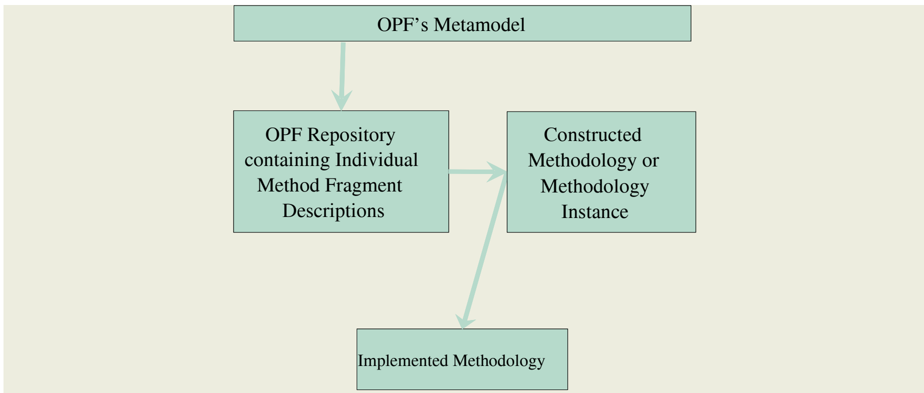


Fig. 3. OPEN defines a framework consisting of a metamodel and a repository of method fragments

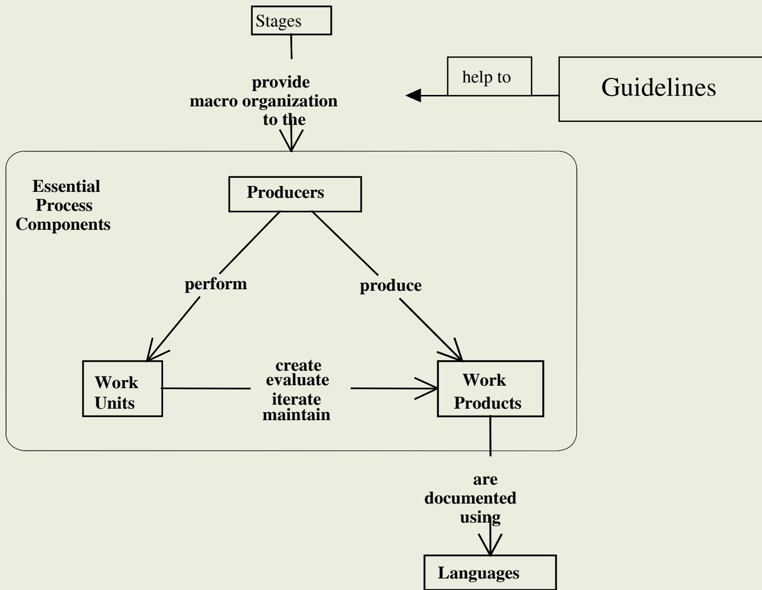


Fig. 4. The five top-level metaclasses of the OPF's metamodel (after Firesmith and Henderson-Sellers, 2002) © Addison-Wesley

"standard" using the supplied construction guidelines (Figure 5) together with a set of deontic matrices (Figure 6). These matrices support the identification of fuzzy relationships between pairs of method fragment types e.g. linkages between tasks and techniques. Deontic values have one of five values ranging from mandatory through optional to forbidden. This gives a high degree of flexibility to the process engineer, perhaps assisted by an automated tool (Nguyen and Henderson-Sellers, 2003), who can allocate appropriate deontic values to any specific pair of process components depending upon the context i.e. the specific project, skills set of the development team etc.

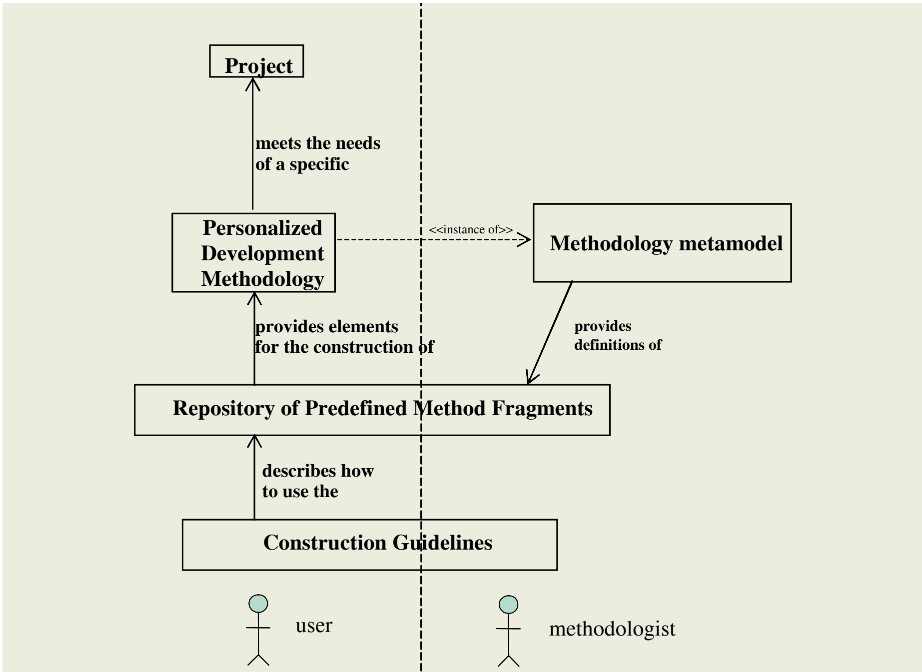


Fig. 5. The methodologist is responsible for the methodology metamodel, creating most of the method fragments in the repository and the guidelines for construction. The user (often the in-house method engineer) uses these guidelines and the contents of the repository (to which they are at liberty to add new fragments) in order to create a "personalized development methodology" attuned to a specific project or context

Tasks

Techniques				
M	D	F	F	F
D	D	F	F	D
D	D	O	O	D
F	O	O	O	F
F	M	O	D	F
R	R	M	R	O
D	R	F	M	O
D	F	M	D	D
R	R	D	R	R
O	D	O	O	R
F	M	O	F	D

5 levels of possibility
M = mandatory
R = recommended
O = optional
D = discouraged
F = forbidden

Fig. 6. One of the deontic matrices is used to link Tasks to Techniques. The values in the matrix represent the likelihood of the occurrence of that pair using five levels of possibility (redrawn from Henderson-Sellers *et al.*, 1998) © Addison-Wesley

Initially, the OPF repository contained about 30 predefined instances of Activity, 160 instances of Task and 200 instances of Techniques (the three main kinds of Work Unit) as well as multiple instances of Role, Stage, Language etc. Some of these are orthogonal to all others in their group and some overlap. Consequently, during process construction both association and integration strategies (Ralyté and Rolland, 2001) are needed. For example, there are several Techniques in the repository for finding objects e.g. textual analysis, use case simulations, CRC card techniques.

As noted above, currently one of the hardest tasks in SME construction is the selection of the optimal set of method fragments to suit any specific situation. Syntactic coupling can be verified in terms of the matching of the output from one fragment to the input for a second. This is facilitated by both the generation of the fragments from a metamodel and also by using a standard way of documenting the fragments (as is done in the OPEN book series, for instance). Nevertheless, the current reality is that the semantic aspect of the fragments must be analyzed "by hand", usually by a skilled method engineer (either in-house or as a visiting consultant or mentor). Work towards a more objective approach is under way (e.g. Nguyen and Henderson-Sellers, 2003; Ralyté, 2004).

Although originally created to support object-oriented software development, several additions have been made to the OPF repository since its first publication in 1997 in order to enhance its support for various new technologies, including additions of relevance to agent technology. In a series of papers, summarized in Henderson-Sellers (2005), we have proposed 39 new Tasks and Subtasks, 23 new Techniques and 28 new Work Products as well as a single new Activity. The method fragments, listed by name only in Table 1, thus provide a significant step in creating a fully supportive AO methodology applicable to a wide variety of types of agent-oriented software development approaches.

It should be noted that of these newly added method fragments, there are a number in common to several of the analyzed AO methodologies. For example, the Task "Construct the agent model" is, naturally, common. Prometheus tends to focus on providing extensions to an OO approach. Consequently, some of the diagrams supported in Prometheus (Pagdham and Winikoff, 2002a,b) can be viewed as UML extensions. Tropos (Bresciani *et al.*, 2004), on the other hand, strive to avoid mere OO extensions and use the AO paradigm explicitly in their modelling of the methodology itself. This introduces some novel diagrams and tasks, which focus on capabilities, as well as on goals and plans. Their focus on early requirements also leads to the need to add a new Activity instance, that of Early Requirements Engineering, to the OPEN repository in order that users of OPEN can re-create the Tropos approach to AO systems development. Gaia (Wooldridge *et al.*, 2000; Zambonelli *et al.*, 2003) is more interested in providing supporting for organizational and social interaction aspects of agents – as is Cassiopeia (Collinot *et al*, 1996; Collinot and Drogoul, 1998) and, to a significant extent, Tropos. This leads to the modelling of responsibilities and permissions as well as the specification of organizational rules, roles, structure and behaviour.

Creation of a project-specific or organization-specific agent-oriented methodology then proceeds using the specifically agent-oriented method fragments listed in Table 1 (which tend to focus only on areas *different* from object-oriented approaches) together with a number of non-agent-oriented method fragments that are needed for those elements of software development that are not technology/paradigm-dependent. These include method fragments to describe project management, some metrics, reusability and so on. A fully comprehensive methodology, suitable for direct industry usage, can be constructed in this way; alternatively, one of the existing AO methodologies can be reconstructed by using only those specific AO fragments. For instance, Henderson-Sellers (2005) shows in more detail how a version of the Prometheus methodology enhanced with some Tropos concepts can be put together from the method fragments

in this newly enhanced OPF repository. Figure 7 shows a portion of the Task-Technique matrix enacted (from Figure 6) for this case study. This shows one way of constructing these matrices. Candidate Techniques (in this example) have been

Table 1. Summary of (a) new Tasks, (b) new Techniques and (c) new Work Products so far added to OPEN in the creation of Agent OPEN. Source documents referred to are: 1. Debenham and Henderson-Sellers (2003), 2. Henderson-Sellers and Debenham (2003), 3. Henderson-Sellers *et al.* (2004a), 4. Henderson-Sellers *et al.* (2004c), 5. Tran *et al.* (2004a), 6. Henderson-Sellers *et al.* (2004b), 7. Henderson-Sellers *et al.* (2004d), 8. Tran *et al.* (2004c), 9. Henderson-Sellers *et al.* (2004e) and 10. Gonzalez-Perez *et al.* (2004)

(a) New Tasks and (indented) associated subtasks	**Refs**
Construct agent conversations	5
Construct the agent model	4, 5, 6, 7
Define ontologies	9
Design agent internal structure	4, 8, 9
Define actuator module	9
Design perceptor module	9
Determine agent communication protocol	1
Determine agent interaction protocol	1
Determine control architecture	1
Determine delegation strategy	1
Determine reasoning strategies for agents	1
Determine security policy for agents	1
Determine system operation	1
Gather performance knowledge	1
Identify emergent behaviour	1
Identify system behaviours	7
Identify system organization	1
Define organizational rules	6
Define organizational structures	6
Determine agents' organizational behaviours	7
Determine agents' organizational roles	7
Identify sub-organizations	6
Model actors	3
Model agent knowledge	8
Model agent relationships	8
Model agents' roles	1
Model responsibilities	6
Model permissions	6
Model capabilities for actors	3
Model dependencies for actors and goals	3
Model goals	3
Model plans	3
Model the agent's environment	1
Model environmental resources	6
Model events	4
Model percepts	4
Specify shared data objects	4
Undertake agent personalization	1
Subtask to Create a System Architecture:	
Determine MAS infrastructure facilities	8, 9

Table 1. (Continued)

(b) New Techniques	**Ref**	**New Techniques**	**Ref**
Activity scheduling	1	Environmental evaluation	2
Agent delegation strategies	1	Environmental resources modelling	6
Agent internal design	4, 5	FIPA KIF compliant language	2
AND/OR decomposition	3	Learning strategies for agents	1
Belief revision of agents	1	Market mechanisms	1
Capabilities identification & analy-	3	Means-end analysis	3
sis		Organizational rules specification	6
Commitment management	1	Organizational structure specification	6
Contract nets	1	Performance evaluation	1
Contributions analysis	3	Reactive reasoning: ECA rules	1
Control architecture	1	Task selection by agents	1
Deliberative reasoning: Plans	1	3-layer BDI model	2

(c) New Work Products	**Ref**	**New Work Products**	**Ref**
Agent acquaintance diagram	4, 6	Network design model	8
Agent class card	8	Platform design model	8
Agent design model	8	Protocol schema	4, 6
Agent overview diagram	4	PSM specification	8
Agent structure diagram	4	Role diagram	5
CAMLE behaviour diagram	10	Role schema	6
CAMLE scenario diagram	10	Service table	6
Caste collaboration diagram Caste	10	Task hierarchy diagram	8
diagram	10	Task knowledge specification	8
Coupling Graph	7	Task textual description	8
Domain knowledge ontology	8	(Tropos) Actor Diagram	3
Functionality descriptor	4	(Tropos) Capability Diagram	3
Goal hierarchy diagram	5	(Tropos) Goal Diagram	3
Inference diagram	8	(Tropos) Plan Diagram	3

Technique	**Tasks** 1	2	3	4	5	6
Abstract class identification						
Agent internal design			Y			
AND/OR decomposition	Y					
Class naming	Y	Y				
Control architecture		Y				
Context modelling	Y			Y		
Delegation analysis	Y	Y				
Event modelling				Y		
Intelligent agent identification		Y				
Means-end analysis ·	Y					
Role modelling	Y	Y			Y	Y
State modelling		Y				
Textual analysis	Y	Y				
3-layer BDI model		Y	Y			

Key:
1. Model dependencies for actors and goals; 2. Construct the agent model;
3. Design agent internal structure; 4. Model the agent's environment;
5. Model responsibilities; 6. Model permissions

Fig. 7. A small portion of the matrix linking Tasks and Techniques for the extended Prometheus case study described in detail in Henderson-Sellers (2005)

identified for the pre-selected (at a previous stage) Tasks. Linkage decisions (here just binary) are made either subjectively/experientially or by means of an overall assessment of a number of factors relating to the project. These factors include CMM level, specific skills in the workforce, domain of the project etc. Note that, even if a candidate is chosen, there is no danger in over-selection since, for an unnecessary Technique, the completed deontic matrix will simply exhibit a blank line (as for Technique: Abstract class identification in this small example – first line in Figure 7).

A next stage of the project is to critically analyze each of these proposed method fragments to see if they are really unique, to ensure there are no overlaps and to ensure compatibility with non-AO method fragments already in the OPEN repository.

Overall, the strengths of this SME approach are that the finally constructed methodology is highly attuned to local conditions and the people in the organization. The challenges are to construct the several deontic matrices, ensuring that (a) linkages accord to the local situation and (b) that the interfaces of any pair of method fragments to be "plugged together" are compatible. Both of these can be facilitated by the use of software tools, the former with a process construction tool (see, e.g., Nguyen and Henderson-Sellers, 2003), the latter with a database-supported evaluation tool (McBride, 2004), both of which we have prototyped.

5 Summary

To date, the evolution of AO methodologies has been disparate with many groups worldwide creating individual offerings. These vary in style and, particularly, in heritage and have a specific focus, either in terms of domain, application style or lifecycle coverage. For industry adoption, it is essential that full lifecycle coverage is achieved in a "standardized" way. One way of achieving some degree of standardization yet maintaining full flexibility is through the use of situational method engineering (SME). With this approach, method fragments are created and stored in a repository or methodbase. For an individual application, only a subset of these is then selected from the repository and a project-specific (or sometimes organization-specific) methodology is constructed. Here, we have demonstrated how this might work by using the OPEN approach that already provides a significant coverage of AO method fragments as well as more traditional OO and pre-OO fragments. Those newer fragments supporting AO approaches are summarized here, describing as they do emerging substantial support for AO methodological creation from SME and the OPEN repository. Further work is needed to consolidate the AO contributions to this repository, to check for inter-fragment consistency and to create a full suite of construction guidelines specific for the creation of AO methodologies suitable for industrial use.

Acknowledgements

I wish to thank Dr Cesar Gonzalez-Perez for his useful comments on an earlier draft of this manuscript. This is Contribution number 04/28 of the Centre for Object Technology Applications and Research.

References

Avison, D. and Fitzgerald, G., 2003, Where now for development methodologies, *Comm. ACM,* **46(1)**, 79-82

Bernon, C., Gleizes, M.-P., Picard, G. and Glize, P., 2002, The ADELFE methodology for an intranet system design, *Agent-Oriented Information Systems 2002. Procs.AOIS-2002* (eds. P. Giorgini, Y. Lespérance, G. Wagner and E. Yu), 1-15

Bresciani, P., Giorgini, P., Giunchiglia, F., Mylopolous, J. and Perini, A., 2004, Tropos: an agent-oriented software development methodology*, Autonomous Agents and Multi-Agent Systems,* **8(3),** 203-236

Brinkkemper, S., 1996, Method engineering: engineering of information systems development methods and tools, *Inf. Software Technol.*, **38(4)**, 275-280

Brinkkemper, S., Saeki, M. and Harmsen, F., 1998, Assembly techniques for method engineering, *Procs. CAISE 1998*, Springer Verlag, Berlin, Germany, 381-400.

Burrafato, P. and Cossentino, M., 2002, Designing a multi-agent solution for a bookstore with the PASSI methodology, **in** *Procs. Agent-Oriented Information Systems 2002* (eds. P. Giorgini, Y. Lespérance, G. Wagner and E. Yu), 102-118

Caire, G., Coulier, W., Garijo, F., Gomez, J., Pavon, J., Leal, F., Chainho, P., Kearney, P., Stark, J., Evans, R., Massonet, P., 2001, Agent oriented analysis using MESSAGE/UML, *Agent-Oriented Software Engineering II* (eds. M. Wooldridge, G. Wei and P. Ciancarini), LNCS 2222, Springer Verlag, Berlin, Germany, 119-135

Castro J., Kolp M. and Mylopoulos J., 2002, Towards requirements-driven information systems engineering: the Tropos project, *Information Systems*, **27(6),** 365-389

Cernuzzi, L. and Rossi, G., 2002, On the evaluation of agent oriented methodologies, *Procs. OOPSLA 2002 Workshop on Agent-Oriented* Methodologies, Centre for Object Technology Applications and Research, Sydney, 21-30

Coleman, D., Arnold, P., Bodoff, S., Dollin, C. and Gilchrist, H., 1994, Object-Oriented Development. The Fusion Method, Prentice Hall, Englewood Cliffs, NJ, USA, 313pp

Collier, R., et al., 2003, Beyond prototyping in the factory of agents, in: *Multi-Agent Systems and Applications III,* LNCS 2691, V. Marik, J. Muller and M. Pechoucek, eds., Springer-Verlag, New York, pp. 383-393.

Collier, R., O'Hare, G. and Rooney, C., 2004, A UML-based software engineering methodology for Agent Factory, *Procs. SEKE 2004* (in press).

Collinot, A. Drogoul, A. and Benhamou, P. 1996. Agent oriented design of a soccer robot team. *Procs. Second Intl. Conf. on Multi-Agent Systems (ICMAS'96)*

Collinot, A. and Drogoul, A. 1998. Using the Cassiopeia Method to Design a Soccer Robot Team. *Applied Articial Intelligence (AAI) Journal*, 12, 2-3, 127-147.

Cossentino, M. and Potts, C., 2002, A CASE tool supported methodology for the design of multi-agent systems, *The 2002 International Conference on Software Engineering Research and Practice (SERP'02)*

Dam, K.H. and Winikoff, M., 2004, Comparing agent-oriented methodologies, *Agent-Oriented Systems* (eds. P. Giorgini, B. Henderson-Sellers and M. Winikoff), LNAI 3030, Springer-Verlag, Berlin, 78-93

Debenham, J. and Henderson-Sellers, B., 2003, Designing agent-based process systems - extending the OPEN Process Framework, Chapter VIII in *Intelligent Agent Software Engineering* (ed. V. Plekhanova), Idea Group Inc., Hershey, PA, USA, 160-190

DeLoach, S.A. 1999. Multiagent Systems Engineering: A Methodology and Language for Designing Agent Systems, *Procs AOIS '99.*

Firesmith, D.G. and Henderson-Sellers, B., 2002, *The OPEN Process Framework*, Addison Wesley, Harlow, UK.

Fitzgerald, B., Russo, N.L. and O'Kane, T., 2003, Software development method tailoring at Motorola, *Comm. ACM,* **46(4)**, 65-70.

Gonzalez-Perez, C., Henderson-Sellers, B., Debenham, J., Low, G.C. and Tran, Q.-N.N., 2004, Incorporating elements from CAMLE in the OPEN repository, *Procs. IIP*, Beijing, 21-23 October 2004

Graham, I., Henderson-Sellers, B. and Younessi, H., 1997, *The OPEN Process Specification*, Addison-Wesley.

Henderson-Sellers, B., 1995, Who needs an OO methodology anyway?, *J. Obj.-Oriented Programming*, **8(6),** 6-8

Henderson-Sellers, B., 2003, Method engineering for OO system development, *Comm. ACM***,** **46(10),** 73-78

Henderson-Sellers, B., 2005, Creating a comprehensive agent-oriented methodology - using method engineering and the OPEN metamodel, Chapter 13 in *Agent-Oriented Methodologies* (eds. B. Henderson-Sellers and P. Giorgini), Idea Group Inc., Hershey, PA, USA

Henderson-Sellers, B. and Debenham, J., 2003, Towards OPEN methodological support for agent-oriented systems development, *Procs. First International Conference on Agent-Based Technologies and Systems*, University of Calgary, Canada, 14-24

Henderson-Sellers, B., Simons, A.J.H. and Younessi, H., 1998, *The OPEN Toolbox of Techniques*, Addison-Wesley, UK, 426pp + CD

Henderson-Sellers, B., Giorgini, P. and Bresciani, P., 2003, Evaluating the potential for integrating the OPEN and Tropos metamodels, *Procs. SERP '03* (eds. B. Al-Ani, H.R. Arabnia and Y. Mun), CSREA Press, Las Vegas, USA, 992-995

Henderson-Sellers, B., Giorgini, P. and Bresciani, P., 2004a, Enhancing Agent OPEN with concepts used in the Tropos methodology, *Engineering Societies in the Agents World IV. 4th International Workshop, ESAW' 2003* (eds. A. Omicini, P. Pettra and J. Pitt), LNAI 3071, Springer-Verlag, Berlin, Germany, 328-345

Henderson-Sellers, B., Debenham, J. and Tran, Q.-N.N., 2004b, Adding agent-oriented concepts derived from GAIA to Agent OPEN, *Advanced Information Systems Engineering. 16th International Conference, CAiSE 2004, Riga, Latvia, June 2004 Proceedings* (eds. A. Persson and J. Stirna), LNCS 3084, Springer-Verlag, Berlin, 98-111

Henderson-Sellers, B., Tran, Q.-N.N. and Debenham, J., 2004c, Incorporating elements from the Prometheus agent-oriented methodology in the OPEN Process Framework, *Procs. AOIS@CAiSE2004*, Faculty of Computer Science and Information, Riga Technical University, Latvia, 370-385

Henderson-Sellers, B., Tran, Q.-N.N. and Debenham, J., 2004d, Method engineering, the OPEN Process Framework and Cassiopeia, *Procs. Symposium on Professional Practice in AI,* Toulouse, France, August 22-27 2004, Kluwer

Henderson-Sellers, B., Tran, Q.-N.N., Debenham, J. and Gonzalez-Perez, C., 2004e, Agent-oriented information systems development using OPEN and the Agent Factory, *Procs. ISD 2004,* Vilnius, 9-11 September 2004, Kluwer

Huget, M.-Ph., 2002, Nemo: an agent-oriented software engineering methodology, **in** *Procs. OOPSLA 2002 Workshop on Agent-Oriented Methodologies*, Centre for Object Technology Applications and Research, Sydney, Australia, 43-53

Iglesias, C.A., Garijo, M., Gonzalez, J.C., Velasco, J.R. 1996. A methodological proposal for multiagent systems development extending commonkads. In *Proc. of 10th KAW*, Banff, Canada

Iglesias, C.A., Garijo, M., Gonzalez, J.C., Velasco, J.R. 1998. Analysis and Design of Multi-Agent Systems using MAS-CommonKADS. In *Intelligent Agents IV: Agent Theories, Architectures, and Languages* (LNAI Volume 1365) (eds. M.P. Singh, A. Rao and M.J. Wooldridge), Springer-Verlag: Berlin, Germany.

Kendall, E.A. and Zhao, L., 1998, Capturing and Structuring Goals, *Workshop on Use Case Patterns, Object Oriented Programming Systems Languages and Architectures*.

Kendall, E.A., Malkoun, M.T. and Jiang, C., 1996, A methodology for developing agent based systems for enterprise integration, **in** *Modelling and Methodologies for Enterprise Integration* (eds. P. Bernus and L. Nemes), Chapman and Hall

Kinny, D., Georgeff, M. and Rao, A., 1996, A methodology and modelling techniques for systems of BDI agents, Technical Note 58, Australian Artificial Intelligence Institute, also published *in Agents Breaking Away: Procs. 7th European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW'96)*, 56-71

Klooster, M., Brinkkemper, S., Harmsen, F. and Wijers, G., 1997, Intranet facilitated knowledge management: A theory and tool for defining situational methods. *Procs. CAISE 1997*, Springer Verlag, Berlin, Germany, 303-317

Kruchten, Ph., 1999, *The Rational Unified Process. An Introduction*, Addison-Wesley, Reading, MA, USA

Kumar, K. and Welke, R.J., 1992, Method engineering: a proposal for situation-specific methodology construction, **in** *Systems Analysis and Design: A Research Agenda,* (eds. W.W. Cotterman and J.A. Senn), John Wiley and Sons, New York, NY, USA, 257-269

Lind, J. 1999. *Iterative Software Engineering for Multiagent Systems. The MASSIVE Method*, LNAI 1994, Springer-Verlag, Berlin

Luck M., Ashri, R. and D'Inverno, M., 2004, *Agent-Based Software Development*, Artech House, Boston, 208pp

McBride, T., 2004, Standards need more rigour, *Information Age*, **Oct/Nov 2004**, 65-66

Nguyen, V.P. and Henderson-Sellers, B., 2003, OPENPC: a tool to automate aspects of method engineering, *Procs. ICSSEA 2003*. Paris, France, **Volume 5,** 7pp

Odell, J., Van Dyke Parunak, H. and Bauer, B., 2000, Extending UML for agents. In G. Wagner, Y. Lesperance and E. Yu (eds.), *Procs. Agent-Oriented Information Systems Workshop,* 17th National Conference on Artificial Intelligence (pp. 3-17). Austin, TX, USA.

Omicini, A., 2000, SODA: Societies and Infrastructures in the analysis and design of agent-based systems, *Procs. First Int. Workshop on Agent-Oriented Software Engineering*

Padgham, L. and Winikoff, M., 2002a, Prometheus: A Methodology for Developing Intelligent Agents. *Procs. Third International Workshop on Agent-Oriented Software Engineering*, at AAMAS'02.

Padgham, L. and Winikoff, M., 2002b, Prometheus: A Pragmatic Methodology for Engineering Intelligent Agents. **In** *Procs.Workshop on Agent-oriented Methodologies at OOPSLA 2002*, November 4, 2002, Seattle.

Pavón, J., Gomez-Sanz, J. and Fuentes, R., 2005, The INGENIAS methodology and tools, Chapter 4, *Agent-Oriented* Methodologies (eds. B. Henderson-Sellers and P. Giorgini), Idea Group Inc., Hershey, PA, USA

Perini A., Bresciani P., Giorgini P., Giunchiglia G. and Mylopoulos J., 2001, A knowledge level software engineering methodology for agent oriented programming, In J.~P. Müller, E. Andre, S. Sen, and C. Frasson, editors, *Proceedings of the Fifth International Conference on Autonomous Agents*, May 2001, Montreal, Canada

Ralyté, J., 2004, Towards situational methods for information systems development: engineering reusable method chunks, *Procs. 13th Int. Conf. on Information Systems Development. Advances in Theory, Practice and Education* (eds. O. Vasilecas, A. Caplinskas, W. Wojtkowski, W.G. Wojtkowski, J. Zupancic and S. Wrycza), Vilnius Gediminas Technical University, Vilnius, Lithuania, 271-282

Ralyté, J. and Rolland, C., 2001, An assembly process model for method engineering, *Advanced Information Systems Engineering*), LNCS2068, Springer-Verlag, Berlin, 267-283

Ralyté, J., Deneckère, R and Rolland, C., 2003, Towards a generic model for situational method engineering, *CAiSE2003* (ed. M.M.J. Eder), LNCS 2681, Springer-Verlag, Berlin, 95-110

Ralyté, J., Rolland, C. and Deneckère, R., 2004, Towards a meta-tool for change-centric method engineering: a typology of generic operators, *CAiSE2004* (eds. A. Persson and J. Stirna), LNCS 3084, Springer-Verlag, Berlin, 202-218

Rao, A.S. and Georgeff, M.P., 1995, BDI agents: from theory to practice. In *Procs. First International Conference on Multi-Agent Systems,* San Francisco, CA, USA, 312-319

Rolland, C. and Prakash, N., 1996, A proposal for context-specific method engineering, *Procs. IFIP WG8.1 Conf. on Method Engineering,* Chapman and Hall, 191-208

Rolland, C., Prakash, N. and Benjamen, A., 1999, A multi-model view of process modelling, *Requirements Eng. J.*, **4(4),** 169-187

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. and Lorensen, W., 1991, *Object-Oriented Modeling and Design*, Prentice-Hall, Englewood Cliffs, NJ, USA

Saeki, M., 2003, CAME: the first step to automated software engineering, *Process Engineering for Object-Oriented and Component-Based Development. Procs. OOPSLA 2003 Workshop*, Centre for Object Technology Applications and Research, Sydney, Australia, 7-18

Schreiber, A. Th. Wielinga, B.J., de Hoog, R. Akkermans, J.M and Van de Velde, W., 1994. CommonKADS: A comprehensive methodology for KBS development. *IEEE Expert,* 9(6): 28-37

Shan, L. and H. Zhu, 2004. *CAMLE: A Caste-Centric Agent-Oriented Modeling Language and Environment*. In *Third International Workshop on Software Engineering for Large-Scale Multi-Agent Systems*. Edinburgh, 24-25 May 2004. [in press]. Springer-Verlag.

Silva, V. and Lucena, C., 2004, From a conceptual framework for agents and objects to a multi-agent system modeling language, *Autonomous Agents and Multi-Agent Systems,* **9(1-2),** 145-189

Sturm, A. and Shehory, O., 2004, A framework for evaluating agent-oriented methodologies, *Agent-Oriented Systems* (eds. P. Giorgini, B. Henderson-Sellers and M. Winikoff), LNAI 3030, Springer-Verlag, Berlin, 94-109

Ter Hofstede, A.H.M. and Verhoef, T.F., 1997, On the feasibility of situational method engineering, *Information Systems*, **22**, 401-422

Tran, Q.-N.N. and Low, G.C., 2005, Comparison of methodologies, Chapter 12 **in** *Agent-Oriented Methodologies* (eds. B. Henderson-Sellers and P. Giorgini), Idea Group Inc., Hershey, PA, USA

Tran, Q.N., Low, G. and Williams, M.A., 2003, A feature analysis framework for evaluating multi-agent system development methodologies, **in**. *Foundations of Intelligent Systems – Procs. 14th Int. Symposium on Methodologies for Intelligent Systems ISMIS'03* (eds. N. Zhong, Z.W. Ras, S. Tsumoto and E. Suzuki), 613-617.

Tran, Q.-N.N., Henderson-Sellers, B. and Debenham, J. 2004a, Incorporating the elements of the MASE methodology into Agent OPEN, *Procs. ICEIS2004 - Sixth International Conference on Enterprise Information Systems* (eds. I. Seruca, J. Cordeiro, S. Hammoudi and J. Filipe), INSTICC Press, **Volume 4**, 380-388

Tran, Q.-N.N., Low, G. and Williams, M.-A., 2004b, A preliminary comparative feature analysis of multi-agent systems development methodologies, *Procs. AOIS@CAiSE*04*, Faculty of Computer Science and Information, Riga Technical University, Latvia, 386-398

Tran, Q.-N.N., Henderson-Sellers, B., Debenham, J. and Gonzalez-Perez, C., 2004c, MAS-CommonKADS and the OPEN method engineering approach, submitted for publication

van Slooten, K. and Hodes, B., 1996, Characterizing IS development projects, **in** *Proceedings of the IFIP TC8 Working Conference on Method Engineering: Principles of method construction and tool support* (eds. S. Brinkkemper, K. Lyytinen, R. Welke) Chapman&Hall, Great Britain, 29-44

Wagner, G., 2003, The Agent-Object Relationship metamodel: towards a unified view of state and behaviour, *Inf. Systems*, **28(5)**, 475-504

Wagner, G., 2004, AOR modelling and simulation: towards a general architecture for agent-based discrete event simulation, *Agent-Oriented Information Systems* (eds. P. Giorgini, B. Henderson-Sellers and M. Winikoff), LNAI 3030, Springer-Verlag, Berlin, 174-188

Wagner, G. and Taveter, K., 2005, Towards radical agent-oriented software engineering processes based on AOR modelling, Chapter 10 in *Agent-Oriented Methodologies* (eds. B. Henderson-Sellers and P. Giorgini), Idea Group Inc., Hershey, PA, USA

Wistrand, K. and Karlsson, F., 2004, Method components – rationale revealed, *CAiSE2004* (eds. A. Persson and J. Stirna), LNCS 3084, Springer-Verlag, Berlin, 189-201

Wood, M. and DeLoach, S.A. 2000, An Overview of the Multiagent Systems Engineering Methodology. *Procs. 1st International Workshop on Agent-Oriented Software Engineering (AOSE-2000)*, 207-222

Wooldridge, M., Jennings, N.R. and Kinny, D., 2000, The Gaia methodology for agent-oriented analysis and design, *J. Autonomous Agents and Multi-Agent Systems*, **3,** 285-312.

Yu, E., 1995, *Modelling Strategic Relationships for Process Reengineering*, PhD, University of Toronto, Department of Computer Science

Zambonelli, F., Jennings, N. and Wooldridge, M., 2003, Developing multiagent systems: the Gaia methodology, *ACM Transaction on Software Engineering and Methodology*, **12(3),** 317-370

MASUP: An Agent-Oriented Modeling Process for Information Systems

Ricardo Melo Bastos and Marcelo Blois Ribeiro

Pontifical Catholic University of Rio Grande do Sul
Av. Ipiranga 6681, Prédio 30, bloco 4, Porto Alegre / RS, 90619-900, Brazil
{bastos,blois}@inf.pucrs.br

Abstract. Multi-agent systems modeling is a very demanding task especially because the software development processes currently being used are based on different paradigms. This work proposes a process to specify agent-oriented information systems that extends RUP. The models are specified through successive refinements using use cases as the reference to express the system requirements. The design model takes into account the infrastructure services required to implement the solution using a multi-agent approach. The models are represented basically by extended UML and AUML diagrams and notations.

1 Introduction

The application of multi-agent systems (MAS) in information systems represents an alternative to solve business processes problems, which requires both decentralization and distribution in the decision-making and execution processes. However, there are no consolidated modeling methods, languages and tools to address these problems.

According to [2], the application of the agent technology in information systems is justified by the following characteristics:

- The domain involves intrinsic distribution of data, problem-solving capabilities and responsibilities;
- It is necessary to maintain the autonomy of the subparts, without losing the organizational structure;
- The interactions are complex, including negotiation, information sharing and coordination;
- The problem solution cannot be completely described a priori, due to the possibility of real-time perturbations in the environment (e.g. equipment failures), and the natural dynamics of the business process.

An approach to modeling information systems requires models that represent their static, functional and dynamic views. For MAS development, it is also necessary to consider two abstraction levels: micro (agent) and macro (societal). At the micro level are expressed the agent architecture and its internal behavior. The macro level represents the agents' society defining the purpose of the system and its requirements.

Information systems are defined according to the requirements identified in the enterprise business processes. A business process involves the participation of actors that interact within the activities to produce the expected results (products, reports,

R. Choren et al. (Eds.): SELMAS 2004, LNCS 3390, pp. 19–35, 2005.
© Springer-Verlag Berlin Heidelberg 2005

business documents, etc) to the enterprise. These activities are executed according to a specific workflow. The decision-making autonomy of the involved actors is limited by the acceptable alternative flows defined for each business process.

This work proposes a modeling process for agent-oriented systems named MASUP – *Multi-Agent Systems Unified Process.* This process is an extension of the Rational Unified Process – RUP [1], which is a consolidate methodology for object-oriented system design. The specification of the system is composed by a set of models derived from UML and AUML [4] [7]. Such as RUP, the artifacts that compose each model are refined in order to produce the next model towards the complete system specification. MASUP is being developed and applied academically since 2002. The system example presented in this work to illustrate the MASUP application was implemented using Java. The agents were thought as an aggregation of objects that ran in a multi-threading environment.

In this work, the requirements, analysis and design workflow activities are presented in detail. Due to size restrictions, the other workflow activities (implementation, test and deployment) were left out of the scope of this work. However, the specification of agents' attributions and interfaces defined in the analysis and the design workflows are important to model and implement the agents, as well as to realize the test of the system.

Section 2 briefly presents a MAS modeled using the MASUP process. Section 3 describes the requirements, analysis and design workflow activities for the modeling process. Section 4 presents the related work. Section 5 presents current and future work. Finally, conclusions are drawing in section 6.

2 The Illustrative Example

This section presents the example used to illustrate MASUP. The example shows a characteristic decentralized planning problem present in industrial production plants. We will explain the concepts necessary for the problem domain understanding in order to enhance the readers experience in the rest of the paper.

A production system involves products, material consumed, components or subassemblies used in the products, production operations and resources to these operations. During the production process, the material is consumed and the resources are used to produce the goods. The resources needed are indicated by a set of manufacturing routines for each product, formed by different production operations. The resources are essentially machines, equipment and labor available in the production system and they have a limited capability.

Considering that production activities consume resources, the resource allocation process is essential to achieve the better balance between cost and benefit for the whole production plant. Production planning is a decision-making activity which involves scheduling algorithms and cost-benefit analysis to specify manufacturing calendars. In a manufacturing calendars is defined which activity must be performed at each instant of time for every resource.

According to [8], the conventional production systems use centralized, global and sequential planning models. In order to optimize the system, the decisions involved in the planning activity are processed in a centralized way. This could lead to a combinatorial explosion due to the number of interactive entities that must be jointly pro-

cessed. Otherwise, the centralized processing determines the global planning of the production system. Any possible variation that could affect the particular planning of one involved execution entity turns the global plan inconsistent, requiring a new planning execution.

To develop a production planning process, a production planner must basically consider:

(i) each item's production route – a set of activities necessary to manufacture the item, showing the precedence relationship which must be observed when executing them;
(ii) necessary resources – some resources have to be used in each activity executed in the process;
(iii) resource capability limitations;
(iv) execution time for each activity – the resources execution time for each activity type must be taken into account for planning their allocation over time;
(v) resources' set-up time;
(vi) the deadlines for each item.

Having these aspects in mind, the question to be answered at each variation on demand must be "is there manufacturing capability to satisfy new demands for the company production plant"? Since every production activity consumes resources, the answer to this question must consider if there are resources available to be used on the production process for these products.

The complexity in the production planning decision-making process is on searching an ideal level of resource utilization, which implies fundamentally on searching the best resources allocation at each moment aiming to satisfy the existing demands. The centralized process clearly has difficulties in considering re-planning in execution time.

Decentralizing the planning process could help to address this issue. The planning problem becomes a distributed process in which each participant decides where and when the local resources will be used. This perspective is different in comparison with the conventional scheduling algorithms, requiring a distinct computational model for its development.

We propose the application of the MAS to the decentralized production planning problem. The idea is to distribute the decision-making among agents that represent the resources and the activities needed to produce a certain item. The reference model used to describe the conceptual modeling of the system is derived from the CIMOSA-*Computer Integrated Manufacturing Open Systems Architecture* framework [12]. At CIMOSA, a Domain Process is functionally decomposed into Enterprise Activities defining a production plan (a logical sequence of Enterprise Activities within a Domain Process). The Enterprise Activities, which are executed by the Functional Entities, correspond to elementary tasks that are normally executed under restrictions. Functional Entities represent the production resources. The production events, such as a production order creation, directly trigger Domain Processes.

Based on the previous ideas a production planning system named M-DRAP – *Multi-Agent Dynamic Resource Allocation Planning* was modeled using MASUP and implemented with Java. M-DRAP proposes an approach to schedule the production orders in real time using software agents and decentralized decision-making.

3 The Multi-agent Systems Unified Process (MASUP)

The Rational Unified Process [1] is one of the most accepted development process in the software industry. RUP is structured in a sequence of workflows and uses UML as modeling language to generate the models constructed in each workflow. MASUP is a RUP variation to model agent-oriented systems. The main goal is to systematically identify the applicability of an agent-oriented solution for the problem. In this sense, MASUP starts like RUP, using the same requirements workflow, but in the analysis and design workflows, it includes different activities and artifacts in order to produce the MAS specification. Fig. 1 shows the MASUP models and the artifacts that composes each model.

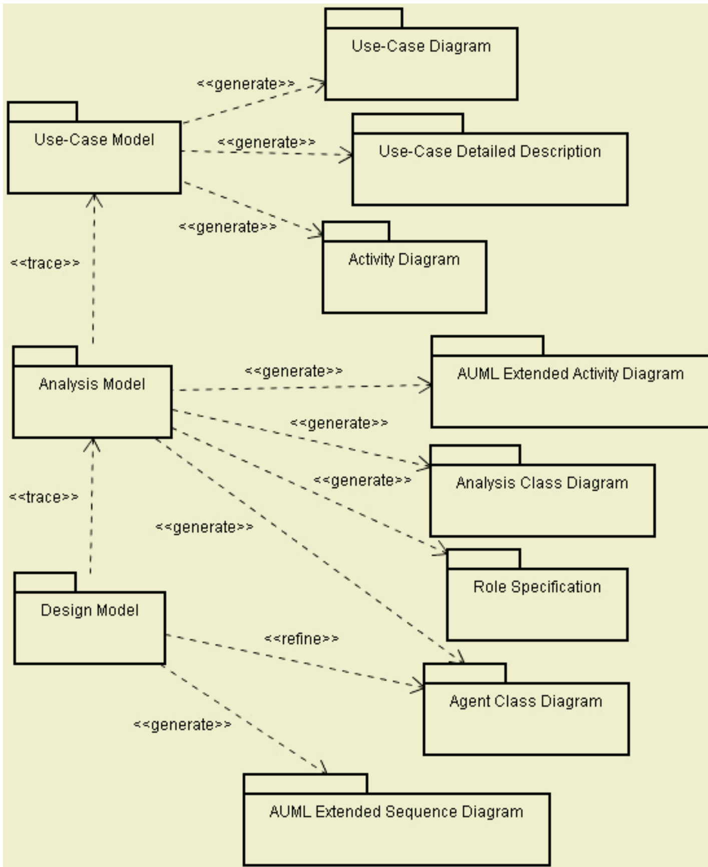


Fig. 1. MASUP models and artifacts

3.1 Requirements Workflow

The RUP proposes a use-case-driven approach to develop information systems. It means that the requirements of the system are captured by the use cases and the system is build to realize them. During the analysis and design workflow activities, the

use-case model produced in the requirements workflow is transformed into a design model via an analysis model. It is possible to apply the same logical process to specify agent-oriented information systems. However, in addition, for a MAS modeling process it is necessary to define procedures to identify the agents and to represent the macro level of the system into each activity.

The requirements workflow captures the user requirements through use cases. The artifacts generated from the requirements workflow are use cases descriptions (textual specification and activity diagrams) and the use-case diagram.

The activities of the requirements workflow are the same proposed by the RUP:

- find actors and use cases;
- prioritize use cases;
- detail the use cases and structure the use-case diagram.

At Fig. 2 is presented a partial use-case diagram for the M-DRAP system. The resource allocation process itself occurs in the use case *Generate Production Order*. Fig. 3 presents the UML Activity Diagram for this use case using the terminology of CIMOSA. The *Generate Production Order* use case involves the allocation of functional entities required to execute the enterprises activities occurrences that compose a domain process. The main problem in this process is the sharing of functional entities among all enterprises activities occurrences in the production system. For each functional entity it is necessary to maintain and update an agenda with its commitments. This characterizes a situation where it is possible to apply a distributed planning process in which every involved functional entity is able to allocate itself to support the production demand, considering the local and global constraints of the production system.

3.2 Analysis Workflow

According to the RUP [1], the purpose of the analysis workflow is to achieve a more precise specification of the requirements, producing a model that represents a first cut of the design. While the use-case model represents an external view of the system, the analysis model describes an internal view of the system. The analysis workflow involves the definition of the architecture of the system, identifying the analysis classes required to realize the use cases. The analysis classes could be related to the activities in the UML Activity Diagram through the objects flow, as presented in Fig. 3.

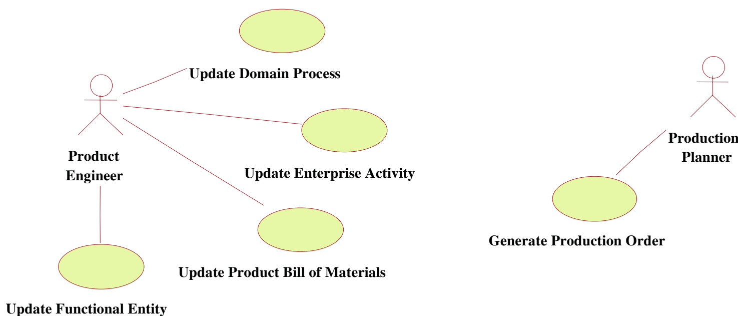


Fig. 2. Partial M-DRAP Use-case Diagram

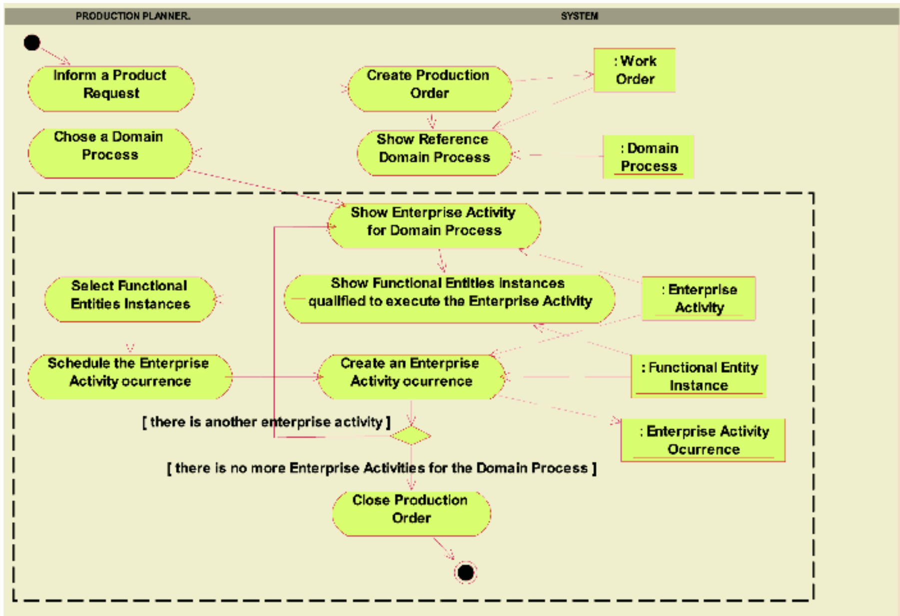


Fig. 3. UML activity diagram for the use case Generate Production Order

Packages are used to organize the analysis model in smaller and more manageable pieces. RUP package identification activity is used in MASUP. MASUP also includes the RUP analysis class diagram creation activity. However, it is necessary to redefine the other activities for agent-oriented solution modeling purpose. After specifying the analysis class diagram and representing the objects required by and generated in each activity, the activity diagrams must be inspected for agent-oriented solution identification. At this point, the system designer may follow the activities flow and analyze if an agent-oriented solution is suitable for the problem. If an agent-oriented solution is appropriate, the analysis workflow must include other activities such as the definition of the agents that compose the society with their attributions and the representation of the interaction among the agents necessary to realize the use cases activities. If an agent-oriented solution is not appropriate, the designers should follow the analysis workflow indicated in RUP.

MASUP analysis workflow main activities are:

- Redesign of the activity diagram to model the agent-oriented solution for the selected use cases;
- Identification of the roles required for the MAS solution based on the activity diagram generated in the previous steps;
- Specification of each agent role defining its attributions;
- Identification of the agents that should play the roles specified;
- Definition of the relationships among the agents composing the agent society architecture.

These activities generate the following artifacts package in the analysis model: AUML Extended Activity Diagram, Roles Specification and Agent Class Diagram.

3.2.1 Redesigning the Activity Diagrams to Model the Agent-Oriented Solution

Considering that some use cases realization present a situation where it is suitable the application of an agent-oriented approach, it is necessary to specify the solution exploring its properties and characteristics. It means to specify the actions related to the application of the agent-oriented solution for the problem.

The use case *Generate Production Order* (Fig. 2) is a candidate to an agent-oriented solution. The agent-oriented solution identification is based on the use case specification shown in terms of the activity diagram presented in Fig. 3. It is possible to notice that the functional entities selection for each enterprise activity is done iteratively in batch by the user. The system does not respond in real-time to ordering variances. This flow also involves problem-solving capabilities based on the functional entities production restrictions and on the overall production demand in the system. If we distribute the decision-making process in order to take into account each functional entity restrictions while maximizing the system throughput, we may include negotiation and coordination capabilities in the system internal elements. Based on these factors and on the items presented in section 1 used to characterize the multi-agent solution appropriateness to a problem, we can continue to model the system using the MASUP workflow activities instead of the RUP ones.

Since an agent-oriented solution is suitable, it is necessary to define a conceptual solution for the problem considering the application of MAS. For this purpose the software engineer needs to identify which activities in the UML Activity Diagram will be played by agents. It includes system activities and actor activities. If an actor activity execution is assigned to agents, the actor transfers its responsibility to the system. This indicates that the agent will have decision-making responsibility in the business process. The activities surrounded by a dashed bold line in Fig. 3 were defined as suitable to have a multi-agent solution in the M-DRAP system.

The software engineer must review the original UML Activity Diagram for the use case and adapt it for the MAS solution. At the M-DRAP system we apply the FIPA-Contract-net protocol [11]. The AUML Extended Activity Diagram for the use case *Generate Production Order* applying the multi-agent solution is presented at Fig. 4. The resource allocation strategy is project-driven and the coordination among the agents is implemented through a market-oriented behavior. We defined a negotiation process, which allows the agents forming coalitions in order to participate in the bidding process. The main objective of the agents is to attend the production demand in an economic, equilibrated and interactive way. Economic, as the proposals must be defined using real production costs. Equilibrated, as the agent always tries to assign the production resource under its responsibility for activities that optimize the resource usage in the production system. Interactive, as the allocation process is executed in real time and in a distributed way involving all the agents qualified to attend a production event.

3.2.2 Identifying Agent Roles from Use Cases Realization

The next step is to identify the roles required to perform the use case activities defined to realize the MAS solution. In MASUP, the analysis objects identified in the RUP analysis workflow are examined in order to identify possible new responsibilities capable to fulfill the roles required to realize the MAS solution. This approach brings as an additional benefit the possibility to establish a relationship among the

analysis objects and the agent roles, which is helpful to integrate the MAS with the rest of the information system in the enterprise. Systems developed under a multi-agent perspective must be considered as part of the enterprise information system.

The following aspects must be considered in order to evaluate the potentiality of an analysis object to be associated to a role:

- Responsibility: the application of the agent-oriented approach in information systems establish decision decentralization and distributed planning of the activities that are executed in order to accomplish the objectives of the system as a whole. Thus, an analysis object could be associated to a role when its responsibilities, under a multi-agent perspective, will require decision-making capabilities and autonomy;
- Information: the attributes maintained by the analysis objects define the knowledge of the system in terms of information. Consequently, this knowledge could be totally or partially relevant to the multi-agent solution. Thus, all the analysis objects that maintain attributes required by the multi-agent solution are natural candidates to be associated to roles;
- Behavior: in the same way of the information knowledge, the behavior of the analysis objects could be total or partially relevant for the multi-agent solution. Thus, the behavior could define an analysis object as a natural candidate to be associated to a role.

Fig. 4 presents the roles compliant with the multi-agent solution identified for the use case *Generate Production Order*. Each request presented by a *domain process* role for an *enterprise activity* role demands one or more *functional entity candidates* roles to be scheduled. When more than one functional entity is necessary to execute an enterprise activity a coalition is created. In this example, it is presented a scenario that requires a coalition formation. It means that the enterprise activity requires at least two functional entities to be executed. The coalition is composed by one or more *coalition members'* roles and one *coalition coordinator* role. Actually, in each coalition the *coalition members* and the *coalition coordinator* roles represent a subset of the *functional entity candidates'* roles selected by the *enterprise activity* as qualified candidates to execute the enterprise activity occurrence. Each coalition candidate for enterprise activity execution must submit a proposal that will be evaluated by the *enterprise activity* role.

Some activities, as for example *Call for candidates to execute functional operations,* require the simultaneously interaction between two roles. In this example, the *enterprise activity* role sends a message to all *functional entity candidate* roles asking about the possibility to participate in the bidding process that involves the execution of an enterprise activity under its responsibility.

The activity diagram shown in Fig. 4 presents some differences from the AUML extended activity diagram. In the AUML, a swimlane identifies each role. A swimlane is an activity diagram partition applied to organize responsibilities for activities. However, this representation does not allow the association of two or more roles with the same activity. Interaction is a basic principle for MAS and involves cooperation, joint planning and negotiation among agents. In order to solve this problem, it is used dependence associations to represent the relationships between the roles and activities. A dependence association between a role and an activity means that the execution of the activity requires the participation of the role.

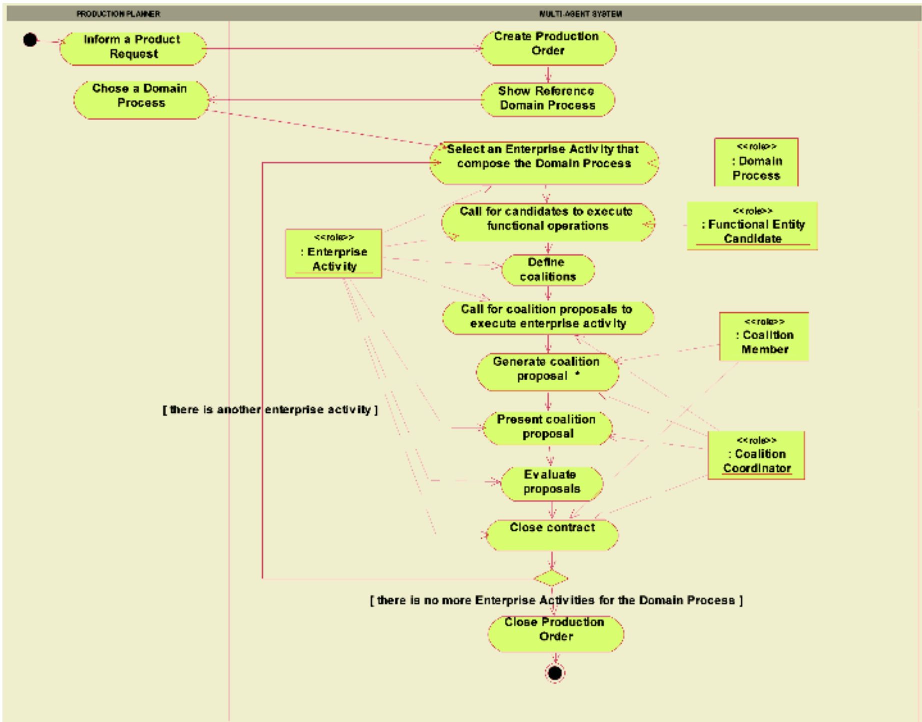


Fig. 4. AUML Extended Activity Diagram for the use case Generate Production Order

3.2.3 Role Specification

The role specification provides the necessary knowledge about the roles required to realize the use cases specified for the MAS. This activity comprehends the identification of the attributions for each role identified in the requirements workflow. The attributions are derived from the use case activities which require the role participation. For each activity could be associated constraints that affect its execution. Fig. 5 shows the Role Specification for the *coalition member* role presented in Fig. 4.

Role: Coalition Member			
Use Case	**Activity**	**Attributions**	**Constraints**
Generate Production Order	Generate coalition proposal	Elaborate and present an individual proposal to the coalition coordinator.	– Keep the commitments previously assumed. – Observe the deadline to send its individual proposal to the coalition coordinator
Generate Production Order	Close contract	Confirm the proposal in order to close the coalition contract.	– Keep the commitments previously assumed.

Fig. 5. Role specification

3.2.4 Identifying Agents

An agent is an aggregation of roles whose attributions are complementary. Complementary attributions mean that the agents should change its role in order to assume another attribution required for a use case activity. According to AUML Extended Activity Diagram presented in Fig. 4, which represents the agent oriented solution defined to schedule a production order in the M-DRAP, the *enterprise activity* role is responsible to recruit functional entities to attend a production demand. The *enterprise activity* role calls for *functional entity* candidates to execute a functional operation. Those *functional activities* candidates form coalitions where there are two roles: *coalition member* and *coalition coordinator*. Actually, both of them just change the role *functional entity candidate* to *coalition member* or *coalition coordinator* when they compose the coalition. Consequently, it is possible to define an agent called functional entity playing three roles: *candidate functional entity*, *coalition member* and *coalition coordinator.* The other roles define a specific agent (domain process and enterprise activity) since these agents do not change their roles.

The architecture of the agent and its behavior is modeled and implemented in the micro level of the system, which is out of the scope of this work. The decision-making autonomy of the agents is limited by the business process workflow.

An agent class specifies the attributions, behavior and architecture shared by a set of agents. Under a macro level point of view, the information required to define an agent class are its name, its maximal number of instances at the society, its attributes, interaction interfaces, its roles and its attributions. The interaction interfaces provide the communication acts that the agent is capable to recognize as valid to attend to a requirement from another agent. The interaction interfaces will be defined just in the design workflow (specifying agent interactions scenarios activity).

Fig. 6 presents the functional entity agent class of the M-DRAP. The agent classes are represented by rectangles where it is defined at least the agent class name and its maximal number of instances. The number of instances specifies the maximum quantity of agent instances at the society.

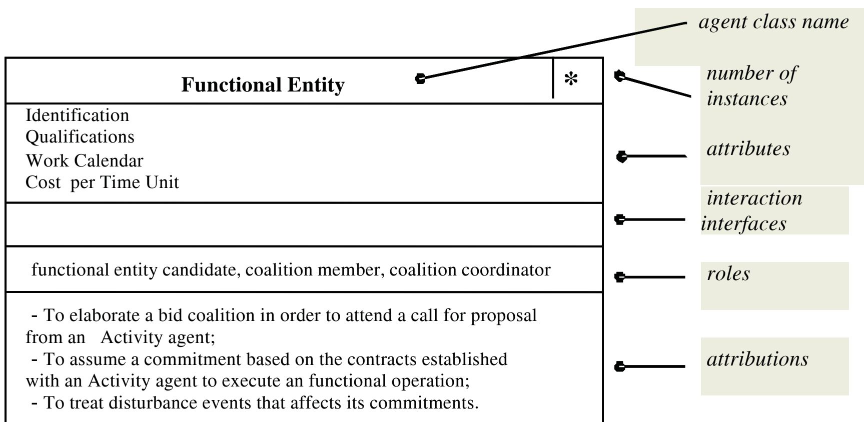


Fig. 6. Agent class specification

3.2.5 Defining the Agent Society

To specify the agent society it is necessary to define the hierarchical relationship between the agents, considering their roles in the activities flow required to realize the use cases of the system. The Agent Class Diagram represents the agent society.

The relationships between the agents are identified from the AUML Extended Activity Diagrams. It means that in the Agent Class Diagram must be defined relationships between agents' roles that are responsible to jointly execute one or more activities. Additionally, according the MAS solution, it is necessary to define the coordination structure for the agent society in order to represent the hierarchical relationship between the agents.

The relationships between the agent classes represent communication channels where messages are exchanged. Arrows connecting both agents' sender and receiver express the relationships between them. The name of these relationships could be optionally identified. The relationship multiplicity and the role of the agent are shown near the end of the path for which it applies. The multiplicity could be exact (1), an interval (1..3), one or more (1..*) or many (0..*). Fig. 7 presents the agent class diagram involving the agents responsible to realize the *Generate Production Order* use case.

In MASUP the hierarchical relationships between the agents are defined as two different types: authority (permanent or role dependent) and communication. In the permanent authority relationship, the receiver must necessarily attend a request from a related agent. A solid line with a bold arrowhead represents this kind of relationship. At the role dependent authority relationships, the receiver answers the request just when the sender is playing a specific role. A solid line with a semi-bold arrowhead represents this relationship. In the communication relationships the receiver locally decides whether it will or not attend a request. A solid line with an arrow represents this kind of relationship.

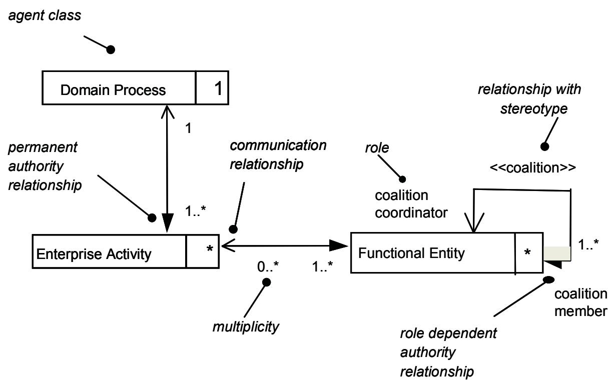


Fig. 7. Agent class diagram

3.3 Design Workflow

The purpose of design workflow is to adapt the analysis results to the constraints imposed by the implementation. In the design workflow MASUP has activities to define the agent interactions and how they will interact with the implementation envi-

ronment. MASUP does not force the use of any specific implementation platform. Because of this, it uses the notion of infrastructure services to map the implementation platform services, showing how the designed agents will interact with them.

The MASUP activities involved in the design workflow are:

- Specification of the agents interaction scenarios;
- Complementation of the agent class specification with the agents communication acts necessary to implement the interactions modeled;
- Identification of the infrastructure services involved on the scenarios specified by the interactions modeled.

The artifacts produced in the design workflow are the Sequence Diagrams and the Agent Class Specification updated by the inclusion of the communication acts necessary to provide the interactions.

3.3.1 Specifying Agent Interactions Scenarios

This activity describes the interaction among the agents that jointly execute a use case activity. The interactions are described by AUML Extended Sequence Diagram [4],[7] including some new properties proposed to allow the identification of agent roles and coalitions, the number of agent instances and the representation of hierarchical relationships between the agent instances. For each activity that composes the use cases of the system, which requires two or more agent for its execution, it is depicted an AUML Extended Sequence Diagram.

The hierarchical relationships defined at the Agent Class Diagram as well as the agent roles attributions specified at the Role Specifications must be observed to depict the Sequence Diagrams in order to guarantee the integrity of the MAS model.

The arrows represent the message exchanged between the agents. Another extension in the AUML Extended Sequence Diagram is the representation of message types. A solid bold line represents a type dependent broadcast message, a message that the sender agent transmits for all the agents that belongs to an agent class. A dashed bold line is used to represent a multicast message that is a message send for a specific set of agents.

Fig. 8 presents an AUML Extended Sequence Diagram for the activity *Call for coalition proposals to execute enterprise activity* (Fig. 4). The messages are expressed by ACL communications acts. The *a:Enterprise Activity* agent sends a call for proposal multicast message to all *Functional Entity* agents, which play the *coalition coordinator* role and belong to a coalition named *production team*.

A dashed line represents the lifeline of the agent and a double line (focus of control) defines its participation in the interaction. The basic flow identifies the regular flow of messages between the agents. The alternative flows express the possible variations in the interactions between agents.

3.3.2 Complementing the Agent Class Specification with the Agents Communication Acts

The AUML Extended Sequence Diagram is the basis for the definition of the communication acts necessary for each agent to run. The interactions modeled must be distributed according to the agent responsible for sending the messages to the other agents. Each ACL message involved in the interaction must be inserted in the agent class specification appropriately. Fig. 9 shows the complete agent class specification

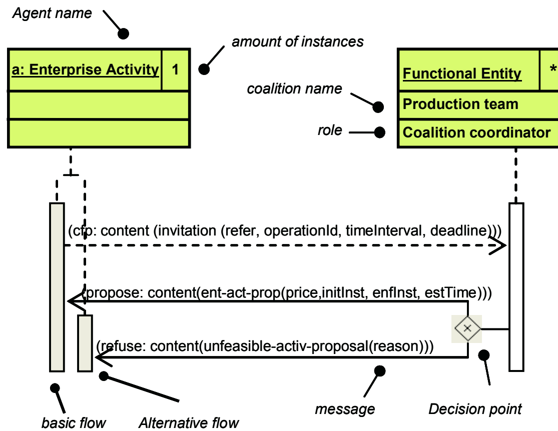


Fig. 8. AUML Extended Sequence Diagram

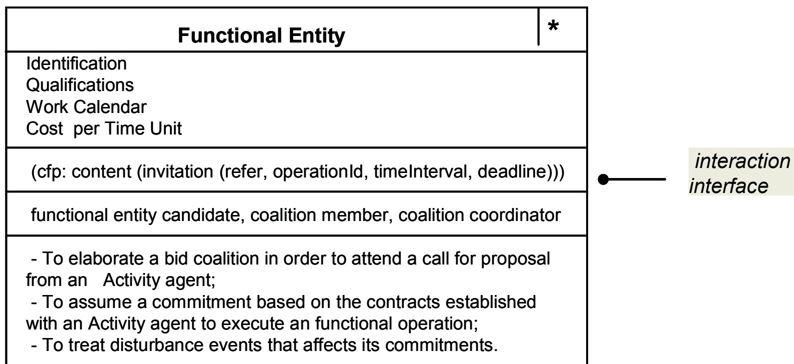


Fig. 9. Agent class specification with communication acts

produced by the addition of the messages needed by the agent to interact with the other agents in the interaction scenarios previously modeled.

3.3.3 Identifying the Infrastructure Services

The design model expresses the particular properties and characteristics of agent-oriented systems. In this sense, some specific services are required to support the operations of the agent society. These services involve the support to the domain agent's in the execution of their attributions, such as accessing the information inside the corporation database and control agent mobility to/from the environment. These services are offered by different agent implementation platforms such as FIPAOS [9] or SemantiCore [5]. Some of these platforms aggregate such services in administrative agents.

MASUP does not impose any implementation platform for the designer. The linkage between the generated diagrams and the implementation platform is done through the indication of which services are necessary in the interaction scenarios previously defined. In order to represent the services needed, the AUML Extended Sequence

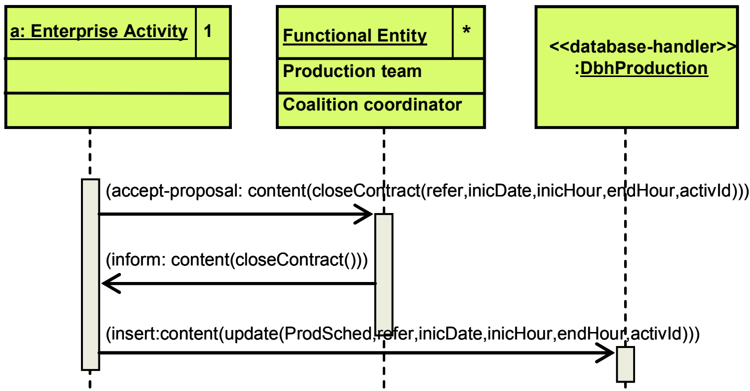


Fig. 10. Design model: AUML Extended Sequence Diagram

Diagram must be revisited and the infrastructure services and interactions with the agents represented.

To illustrate this procedure, Fig. 10 presents an AUML Extended Sequence Diagram for the activity *Close contract* (Fig. 4), where the infrastructure services identified by the stereotype «database_handler» offers an interface that contains a set of queries and data update capabilities that may be requested by the domain agents. The <<database_handler>> Dbhproduction constitutes the service access API provided by the implementation platform. It receives a request to update the ProdSched database including an enterprise activity occurrence and its respective information about execution time. Notice that the message exchange represented between the agent and the service is a simple message instead of an ACL message. The message representation format between the agent and the service is dependable on the implementation platform. If the platform uses Java for example, this message could be a simple method invocation.

4 Related Work

Agent-oriented software development is a growing research area since there are no standards for agent systems design and implementation. In spite of the lack of standards, there are many interesting works showing a methodological approach for agent system development. This section compares three well-known agent-oriented methodologies presented in the literature to MASUP. The objective is to show the MASUP weaknesses and strengths and to explicit MASUP contributions in relation to the state of the art research in the area. An in-depth comparison between MASUP and other methodologies is out of the scope of this paper due to size restrictions.

The MaSE methodology [6] has two main development phases: the analysis and the design. In the analysis phase, MaSE applies use cases to identify a set of roles and communications paths within the system. The use case specification allows the definition of the basic scenarios for the system and depicts Sequence Diagrams for each scenario. The phase finishes with the specification of the roles and tasks related to the system domain. In the design phase, the designer represents the agents, the roles they play and the conversation between them in a single agent class diagram. After detail-

ing the conversation between the agents, the designer finishes the phase by creating the agents internal structure and the system design (a deployment diagram showing the agents location within the system).

MaSE uses different diagrams to capture the different systems views such as MASUP. It applies use cases as the basis for scenarios definition but it does not identify which use case requires an agent-oriented solution as proposed in this work. The MaSE's UML based diagrams have the same limitations UML has to model the differences between OO and agent systems. The MaSE activities and artifacts do not permit to define the aspects related to the hierarchy between the agents, coalition representation and multicast/broadcast message exchange. MaSE do not model the computational environment and the specialized services required to support the agents' activities in MAS. This may lead to a conceptual gap between the design model and the implementation infrastructure.

The Tropos methodology [3] proposes a process with five phases: early requirements, late requirements, architectural design, detailed design, and implementation. Tropos uses a requirement-driven approach for MAS modeling. It elicits actor e goals in the early requirements. The early requirements phase also models the plans and resources used to achieve the goals defined. The following phase, late requirements, details the artifacts previously defined refining the goals in terms of sub-goals. The phase provides a diagram showing goals and sub-goals relations in the system organization. The architectural design phase derives new actors to fulfill non-functional requirements identifies in the previous phase and generates capabilities and agent types based on the actors that group these capabilities. The detailed design phase model the system micro level, defining the agent plans and the agent interactions. These definitions are done through UML activity diagrams and AUML interaction diagrams respectively. Tropos use in the implementation phase the JACK Intelligent Agents [10] platform to implement the agent solution. The design artifacts are mapped to the JACK constructs in the implementation phase.

The Tropos methodology represents an important referential for the definition of requirements in MAS, however the proposed process do not cover important aspects in business processes as agent role change, hierarchy among the agents and coalition formation as explored in MASUP. Tropos also uses a proprietary notation to represent the requirements and a fixed implementation platform. As other methodologies, Tropos consider the macro and micro level, defining different process phases specially used for the architectural (macro) and detailed (micro) system views.

The Prometheus methodology [13] consists of three phases: the system specification, the architectural design and the detailed design. The system specification phase identifies the system functionalities, goals, percepts and actions. Prometheus uses scenarios descriptions to specify the steps necessary for a functionality realization. The architectural design phase is primarily concerned in the identification of the agents in the system and their interaction. The agents defined group functionality and the related percepts, actions and events necessary in the functionality. The main diagram of this phase is the system overview diagram that shows actions, data, events, capabilities, plans, agents, protocols and messages altogether. The detailed design phase maps the agents' internal capabilities with de capability descriptors. It also uses the same notation of the system overview diagram to model agent, capabilities and plans internal components.

Although presumed to have start-to-end support for agent systems modeling, Prometheus does not have a requirements phase. It also describes scenarios, but does not use case descriptions to do so. Many artifacts are textual descriptions of each item. This lacks formality and difficult the traceability between de different levels of abstraction. Prometheus also mix different systems views (static, functional and dynamic) in one diagram which difficult the system understanding.

5 Current and Future Work

Nowadays we are working on the specification of a process for the micro level (agent) compatible with the models presented in this work, considering the integration of both modeling levels (micro and macro level) and implementation platforms. We are using MASUP to develop a multi-agent based resource allocation system for project management. The system uses a negotiation process to allocate the appropriate employees to the project activities based on competences. We are also developing a knowledge management system using MASUP that aims to capture and share knowledge represented in OWL. The system uses the approach of personal agents associated to the company employees to interact with the users.

As future works we intend to extend MASUP and integrate it with some agent implementation platforms such as the SemantiCore. It is possible to map the design artifacts to SemantiCore basic abstractions, such as: agents, action plans, actions, and so on. For instance, the messages represented in the sequence diagram could be mapped to effectors and the ACL speech acts coordinated by the action plan controlling mechanism in the Execution component (which is responsible to coordinate the agent actions in the society).

6 Conclusion

This work proposed a process which extends RUP to develop agent-oriented systems. The models produced in each activity of the process are specified through successive refinements using use cases as the reference to express the system requirements. The design model takes into account the implementation infrastructure required to implement the solution using a multi-agent approach.

The main contribution of this work comprehends a consistent system view based on integrated models guiding the developers from the system requirements to the implementation platform mapping. Other contributions are:

- The process does not direct the designer to the multi-agent solution at the first hand such as other processes presented in the literature. The process indicates where the designers must check if the multi-agent solution is appropriate for the problem.
- RUP is a well accepted software industry process. Extending RUP reduces the learning curve needed to understand a new software development process.
- The models constructed are based on diagrams extended from UML and AUML. This similarity facilitates the comprehension and use of the diagrams.
- The process evolved from previous versions developed in 2001 as a result of their application for multi-agent planning solutions.

References

1. Kruchten, P., The Rational Unified Process – An Introduction, Addison-Wesley, 2000
2. Jennings, N.R. et ali., "Using intelligent agents to manage business processes", Proceedings of Practical Applications of Intelligent Agents and Multi-Agent Technology – PAAM'96, London, UK, 1996.
3. Castro, J., Kolp, M., Mylopoulos, J., "Towards Requirement-Driven Information Systems Engineering: The Tropos Project", Information Systems, 27, (2002) 365-389
4. Odell, J., Parunak, H.V.D. Bauer, B., "Extending UML for Agents", Proceedings of the Agent-Oriented Information Systems Workshop at the 17th National Conference on Artificial Intelligence, Austin, TX, (2000) 3-17
5. Blois, M., Lucena, C., "Multi-Agent Systems and The Semantic Web - The SemanticCore Agent-based Abstraction Layer", Proceedings of the 2003 ICEIS - International Conference on Enterprise Information Systems, Porto, 2003.
6. DeLoach, S.A., "Multiagent Systems Engineering", International Journal of Software and Knowledge Engineering, vol. 11, no. 3, (2001) 231-258
7. Bauer, B., Müller, J.P., Odell, J., "Agent UML: A Formalism for Specifying Multiagent Interaction", Agent-Oriented Software Engineering, Lecture Notes in Computer Science, Springer-Verlag, Berlin, (2001) 91-103
8. Parunak, H.V.D. Applications of distributed artificial intelligence in industry. In: O'Hare, G.M.P.; Jennings, N.R. (Eds.). Foundations of distributed artificial intelligence. New York: John Wiley & Sons, 1996.
9. Poslad, S., Buckle, P., and Hadingham, R. "FIPA-OS: the FIPA agent Platform available as Open Source", Proceedings of the Fifth International Conference on the Practical Application of Intelligent Agents and Multi-agent Technology, PAAM, Manchester, 2000, pp. 355-368.
10. Busetta, P., R¨onnquist, R, Hodgson, A. and Lucas, A. JACK Intelligent Agents - Components for Intelligent Agents in Java. Technical report, Agent Oriented Software Pty. Ltd, Melbourne, Australia, 1998.
11. FIPA Communicative Act Library Specification. Disponível em http://www.fipa.org/specs/fipa00037/
12. Zelm, M. et ali., "The CIMOSA business modelling process", Computers in Industry, 27, p.123-142, 1995.
13. Winikoff, M., Padgham, L. "Prometheus: A Methodology for Developing Intelligent Agents (2002)", Proceedings of the Third International Workshop on Agent-Oriented Software Engineering, AAMAS, 2002.

Composition of a New Process to Meet Agile Needs Using Method Engineering

Massimo Cossentino and Valeria Seidita

Istituto di Calcolo e Reti ad Alte Prestazioni,
Consiglio Nazionale delle Ricerche Viale delle Scienze, 90128 Palermo, Italy
{cossentino,seidita}@pa.icar.cnr.it

Abstract. The need of developing a new software engineering process to allow the quick prototyping of some robotic applications and meet the requests by some companies for a development process that was shorter than PASSI, gave us the opportunity of applying our studies on the assembling of a new SEP by reusing parts (called method fragments) from other processes. In this paper we discuss our approach that, starting from the method engineering paradigm, adapts and extends it considering specific agent-oriented issues like the multi-agent system meta-model. The final result of our experiment (Agile PASSI) is presented together with the requirements that motivated its structure.

1 Introduction

Many different design methodologies for multi-agent systems can already be found in literature and nonetheless further works propose brand new approaches or the extensions of existing ones. We think (but the opinion is largely shared in the scientific community) this happens because each methodology has been conceived to solve a specific problem in a fixed context and this strongly limits the possibility of reusing it (without significant changes) in a different situation. Several developers respond to their need of designing a specific system in some productive context by creating a specific design methodology; this implies a big effort and the cost of developing a MAS (multi-agent system) becomes higher than the comparable object-oriented solution (it is worth to notice that in the object-oriented context the Unified Process is an accepted standard and designers does not need to add the design process construction cost to the development effort).

A new branch of Software Engineering, called Method Engineering [1, 2] proposes to create a new methodology starting from existing methodology parts, called *method fragments*, that a method engineer defines and stores in the method base. When a method engineering wants to design a new methodology, he extracts and assembles the fragments (each one composed of some work to be done, the resulting artifacts and supporting guideline) in order to obtain a methodology that is suitable for his specific needs. Because of the great number of methodologies from which method fragments can be extracted, it is necessary to represent them in a standard way and to have a definition of the method fragments that could fit it. This work consists in a re-engineering process [3] of existing methodologies to identify and extract fragments that could be

R. Choren et al. (Eds.): SELMAS 2004, LNCS 3390, pp. 36–51, 2005.
© Springer-Verlag Berlin Heidelberg 2005

used in the new methodology construction process. We think that in the AOSE (Agent-Oriented Software Engineering) context, some confusion still exists among the use of the terms process, methodology and method. In order to avoid misunderstandings, in this work, from now on, we decided to refer to the (design) process (avoiding the use of the word methodology) meaning with it the collection of phases, activities and steps that produce the project deliverables. The term method will be used with the meaning of a way of performing some kind of activity (at whatever level) within the design process (this includes techniques, artifacts, and guidelines).

As a consequence of this adopted terminology, we will refer to the final result of the method engineering activity as a new *process* or indifferently as SEP (Software Engineering Process). It will be composed by a set of method fragments, each one of them specifying which phase/activities or more generally work definitions should be carried on and by which stakeholders. The most frequent aim of these work definitions is to produce/refine one or more artifacts (text documents, diagrams, ...) and in so doing they often refer to some kind of style template (text documents) or modeling language (diagrams). This process in order to be successfully applicable should be complemented by some guidelines that will help the involved stakeholders in performing their duties according to some defined best practices. The process will also prescribe in which sequence the phases and activities will be executed and if iterations should be done or feedbacks provided to previous items; this often relates to some common models like the waterfall [4] and evolutionary [5] (including iterative and incremental) ones.

Before proceeding to fragments assembling, we need to describe and represent these parts in a standard way so to make easy the composition of parts coming from different processes. The first step of this work consists in the creation of the meta-model that will be used to describe the existing processes and the multi-agent system structure. An important contribution to the solution of the first issue comes from an OMG specification, the Software Process Engineering Metamodel [6]; this is the natural candidate to become the adopted meta-model process, since it is already an accepted standard in the OO context. We have exploited the possibilities offered by SPEM in the specific agent-oriented context obtaining interesting results in the modular representation of PASSI [7] and the method fragment extraction from it, using the standard definition of method fragment [8]). In this paper we will present our approach to the reuse of these fragments for assembling a new process (Agile PASSI) that satisfies our development needs of specific robotic application.

The paper is organized as follow: in the next section we present an introduction to the key topics of this work: method engineering and agile processes; in section 3 we present our general approach to the new process composition and related method fragments selection; in section 4 we quickly present the PASSI process from which we have extracted the method fragments; in section 5 we report the results of our experiment, and finally some conclusions are drawn in section 6.

2 Theoretical Background

In studying the solutions presented in this paper we considered a specific problem, the rapid development of an agent-oriented application that needs a low level of quality in the design and its documentation. This brought us to identify the need for an agile

process that could be supported by some design tool. Taking profit of our previous experience with the PASSI process [9], patterns reuse [10], and related design tools [11], we conceived an agile version of PASSI by reusing some of its parts and building up the new process. This corresponds to apply the method engineering approach that will be discussed in subsection 2.1 to the composition of this process. All of these issues will be discussed in the following sub-sections.

2.1 Agent-Oriented Method Engineering

In order to build our new design process we adopted (and extended) the method engineering paradigm [12][13][14]. According to this approach, the new SEP (Software Engineering Process) is built by assembling pieces of the process (method fragments) [2][1][3] from a repository of methods. In this way we could obtain the best process for our specific needs. We chose this approach because, in the last years, it proved successful in developing many object-oriented applications, for example information systems [15], and is now collecting a growing interest from the agent community[16][17].

Some differences exist between the approach we used in building Agile PASSI and the cited approaches in the object-oriented context; the most relevant one is that the OO context refers to the object concept and related model of the object oriented system, while we refer to a MAS meta-model, that is a structural representation of the elements (agent, role, behavior, ontology,...) that compose the actual system with their composing relationships. We built Agile PASSI by adopting the MAS meta-model represented in Figure 2, that will be presented more in details in sub-section 3.1.

Figure 1 presents what we think to be the correct process for composing a new SEP under the evolution of the method engineering paradigm that we call agent-oriented method engineering. The process begins with the introduction in the method base of the fragments extracted from available processes and the specifically created new ones; then the designer (or better the method engineer), before building the new SEP, identifies the elements composing the meta-model of the kind of MAS he wants to build. The composition of the new SEP is performed under the assistance of some specific software tool, called CAPE (Computer Aided Process Engineering) or CAME (Computer Aided Method Engineering) depending on its process or method-oriented vocation. This tool will allow the selection of the right method fragments from the method base and will permit their introduction in the selected (or specifically designed) process model.

In this process, the definition of the MAS meta-model will help at both a logical and practical level. Firstly this will be useful in the method fragment selection phase (avoiding the selection of methods dealing with elements that are not present in the defined MAS meta-model) and secondly, the same fact of clearly declaring the structure of the system will allow the design tool to check for model coherence and to find not completely defined parts. Once the new SEP has been composed, the same CAPE/CAME tool should permit the instantiation of a simpler tool (a CASE, Computer Aided Software Engineering, tool) that will be used by the designer when designing a system to solve some specific problem.

Agile PASSI has been constructed according to this process and in defining/composing our fragments we used a CAME tool (MetaEdit+ by Metacase) that offered a specific support for the composition of a process from existing fragments.

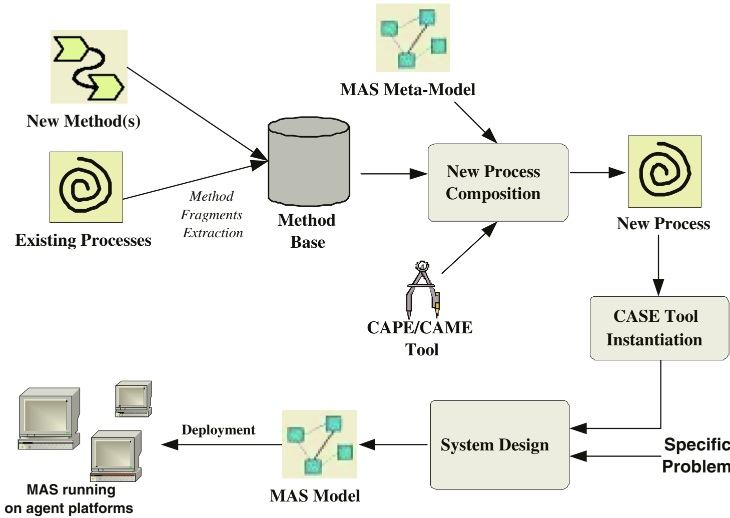


Fig. 1. The adopted Agent-Oriented Method Engineering process.

2.2 Agile Processes

Classic SEP are well disciplined and heavily oriented to make a process predictable and have a great stress on planning. As a reaction to this way of developing a software,in the last years a new kind of processes, called lightweight in a first time but now known as agile, has been developed. An important difference between the two kinds of processes is the smaller quantity of documentation produced in the second case, in fact agile ones are code-oriented, being source code the key element of documentation, and by modeling with little increments and iterations they can easily face changes. In Agile processes the consequence of an iterative development is to realize working subsystem that have not (yet) all the functionalities of the final system, but when integrated and tested, they will provide the requested features. We can say that agile processes (often called agile methodologies) are not complete processes but they are a supplement to the already existing ones, they begin where the other fault or better where the other needs changes in order to perceive their objective. Reexamining PASSI we used principles and techniques of agile processes [18] in order to create a lightweight SEP, simple, easy to use and principally based on code production rather than on documentation (that is still requested, but mostly when it can be automatically produced), besides we considered the sequence of activities defined in one of the most used agile approaches, Extreme Programming [19]: (i) Planning, (ii) Designing, (iii) Coding, and (iv) Testing; this sequence will constitute the center of the proposed SEP.

3 The Proposed Approach

This section proposes our approach to the composition of a new process. In this specific work we will apply our ideas to the reuse of PASSI fragments in order to build

a new process (Agile PASSI) accordingly to some requirements presented later. In our research activity we decided to adopt existing standards whenever possible in order to remain as close as possible to industrial needs in this direction; for this reason we adopt: SPEM (Software Process Engineering Metamodel) by OMG [6] in modeling our processes (and related fragments), UML (extending it when necessary) in modeling our artifacts; FIPA [20] as the reference agent architecture and XML for data representation.

In creating a new process, we consider that this essentially is a design activity by itself and as such it should be ruled by some kind of design process. The design process we adopt (to design a new process) is composed of four phases: requirements analysis, process model design, fragments selection, and fragments integration (it includes the assembly and adjustment activities performed to adapt the fragments to the new process). Further iterations in this sequence of phases should aim at process maturity as described in the CMM [21] but these aspects are out of the scope of this paper.

The **Requirements Analysis** phase, consists of the identification of the important features of the process under construction; for instance the need of an highly detailed level of design that could derive from a defense contract project. Another example could be the indisputable dependability requested to a mission critical system like an avionic one. The **Process Model Design** consists of the selection of the process model (waterfall [4], evolutionary or incremental [5], transformation [22], spiral [23], . . .), the phases that constitute it and other process level requisite (for instance the conditions that enable each new iteration). We consider situational requirements as the most useful guidance for selecting the right process model. The need of rapidly facing changing requirements could bring to the adoption of an evolutionary process while, conversely the need of a very formal development process, with high quality level insurance could lead to the adoption of some IEEE guidance [4][24] and therefore to the selection of a waterfall model. The **Fragments Selection** phase aims at identifing the best fragments for achieving the process goals (according to the requirements identified in the first phase). Some authors (Ralyté et al. [2]) identify method fragments (called 'chunk' in that work) using a process-driven very structured and completed heuristic. We think that this top-down method, although very clear and well defined is not sufficient to meet all the requirements (that are often expressed also in terms of deliverables and architecture of the system to be developed). For this reason we found useful to complement a process driven selection activity with another data-driven one that considers aspects like diagrams/other documents to be produced and the system architecture according to some kind of MAS (multi-agent system) meta-model. From the process-driven point of view, we consider four different levels of method fragment granularity according to the position of the fragment in the process (in this classification we adopt the SPEM terminology): *Phase* (highest level parts of the process, usually characterized by an entry condition, a goal and the sequential constraint, for instance System Requirements and Agent Society in PASSI), *Work Definition* (a substantial part of the operations to be performed in the process, it usually is composed of several lower level elements; for instance Agent Identification, and Domain Ontology Description in PASSI, see section 4), *Activity* (usually the smallest reusable part of a process, an activity is composed by the tasks, operations, and actions that are performed by a role or with which the role may assist, for instance Use Case Identification and Roles Dependencies Analysis in PASSI

[7]), *Step* (the atomic elements that compose an activity, for instance the different steps of the heuristic used for identifying agents from use cases in PASSI).

The selection of method fragments (that in our approach could be at the *Phase*, *Work Definition* or *Activity* level of granularity) is performed working on two dimensions: the process dimension and the system architecture dimension. The *process dimension* enables a zooming on the analysis done during the process model design and considers lower level features of the process. For instance at this stage we evaluate the need for a specific attention on security (from which we will deduce the importance of introducing some specific method fragment). Essentially in this phase we first select the phases we want to introduce in the process (while some process models, like the waterfall one, already prescribe these phases, some others leave a considerable level of degree in this choice), and then we select the lower level fragments inside them. In so doing we follow some criteria:

- Process completeness: all phases (and their activities) of the process are to be covered by appropriate method fragments;
- Process coherence: generally speaking, each fragment refers to some kind of 'philosophical' or 'methodological' approach to the solution of the problem it faces. It makes no sense to introduce fragments belonging to contrasting approaches in the same process;
- Process applicability: the selected fragments should compose a process that is realistic (not too complex or simplistic for the faced problem) and lead to the final solution in an optimal (or at least acceptable) way (in terms of cost and time);
- Contracts accomplishing: each fragment has some specific preconditions that should be enacted by previous parts of the process and when it has been applied, it generates some postconditions that could trigger the following fragments;
- Stakeholders adequacy: it consists in selecting a set of fragments where the skills required for involved roles (analyst, architect, programmer, ...) are adequate to the situation (company, developing team, . . .) where the process will be applied;
- Stakeholders satisfaction: people involved in the process application play a decisive role in the success of the project. Their expectancy in terms of the kind of work they will participate, is an important factor for the selection of fragments.
- Specific requirements: they could help in the selection of some fragments. For instance the need of designing a real-time system will induce to consider fragments that deal with time-related aspects of the design.

In the *system architecture dimension* we define the MAS meta-model and from it we deduce the need for specific fragments that with their resulting artifacts could contribute to the definition of a system obeying to the defined meta-model. We now deduce the models and views that are necessary to define and refine the elements of the system (this in someway resembles the product perspective of Brinkkemper et al. in [3]).

During the **Fragments Integration** phase, the selected fragments are disposed in the right position inside the process and when necessary they are adapted to the new context. Method fragment contracts (preconditions required by each fragment and postconditions enacted by it) are used to verify the possibility of directly connecting some fragments. An interesting approach to the adaptation of fragments is described in [25].

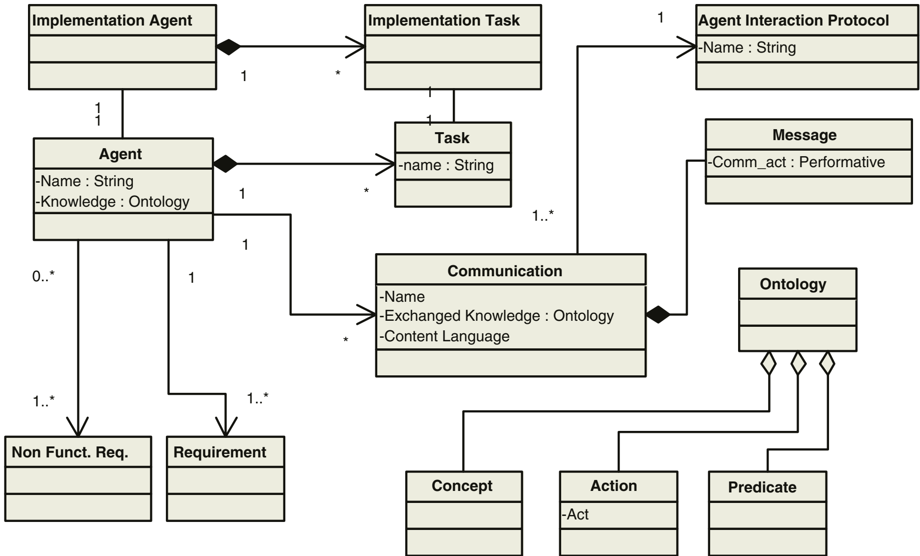


Fig. 2. The Agile PASSI MAS meta-model.

3.1 The Agile PASSI Process Composition Experiment

The reported experiment started from two different motivations, the first was that we needed a short design process to let designers focus on the implementation of relatively small robotic applications; the second motivation was that during the development of large projects some of ours industrial partners underlined the benefit that could come from the availability of a versatile process that could substitute PASSI in the development of minor parts of the whole project. Because of space concerns, in the following we will only refer to the first motivation but the other has been considered too during the Agile PASSI construction and the resulting process proved good in the developing of non robotics applications too. Our robotic systems are deployed on mobile robots moving at a relatively low speed (only a few meters per second) and usually performing missions related to the use of cognitive capabilities (for example we designed systems for museum guide, surveillance and environment discovery applications). We now want to design a process that, taking profit of the successful experience already done with PASSI, could be the best solution for this kind of problems in our laboratory context.

The requirements that we could identify for our new process are centered on the main goal of not distracting developers from their main objective of implementing/ tuning some kind of new algorithm with a long design process; nevertheless, we do still need to maintain a reasonable quality of design documentation for enabling the transferred knowledge among people in our laboratory. Another wish is related to the possibility of quickly reusing contributions coming from other projects in order to restrict the effort related to the development of a new application to the solution of its novelty aspects. For instance, in a robotic application great parts of already existing systems can be reused both from the algorithmic (general navigation solutions like path planning and obstacle avoidance) and structural (communications, resource sharing and

data caching) points of view. As regards the response time of the developed systems, our real-time constraints are not very tight (as already said, ground robots move relatively slow) but nonetheless the possibility of explicitly designing concurrent actions and time relationships among them is highly desirable in order to optimize the performance of a system that because of the use of low efficiency agent platforms (Java-based) could otherwise bring to an unacceptable decay in performance if no specific attention is given to this problem. We think that all of these issues could be satisfied by using an agile process that supports a light (manual) design phase while encourages the reuse of existing contributions in form of patterns and (automatically) produces a consistent documentation at different level of abstractions.

As regards the other dimension we consider in our composition approach (the system architecture), the requirements of the new process regard our decision of significantly reducing the dimension of the conventional PASSI MAS meta-model [26] because of the direct relationship that exists between the number of elements of the meta-model and the design artifacts (and activities). The chosen MAS meta-model is reported in Figure 2, it is composed of four different categories of elements: requirements (functional and non functional requirements), domain ontology (concept, predicate, action), agent logical (abstract) structure (agent, task, communication, message, agent interaction protocol), and agent implementation structure (implementation agent and implementation task).

In this meta-model, the concept of agent represents the entity performing the system functionalities. Each functionality descends from one or more requirements elicited during meetings with clients, users, developers and designers and then represented in a conventional use case diagram. Agent knowledge is described in terms of instances of the domain ontology, that is a composition of concepts (entities and categories of the domain), predicates (assertions about elements of domain) and actions (that agents can perform in the domain, so affecting the status of concepts). In Agile PASSI we think to an agent as composed of tasks representing a portion of its behavior and embodying its capabilities of pursuing a specific goal. An agent uses communications to realize its social relationships and asking for collaborations from other agents. Each communication is composed of messages expressed in an encoding language and refers to an element of the ontology, besides the flow of messages is ruled by an interaction protocol (AIP)

From all of these requirements we deducted some choices for our new process:

- We decided to adopt an agile process. This introduces a specific structure of process model: (a) it should be short (composed of only a few phases), iterative, and incremental (as a consequence we need some iteration planning activities) and (b) a specific attention is devoted to coding and testing in order to have a frequent delivery of functional portions of the final system; this solves the developer 'anxiety' of focusing on algorithm implementation rather than system design.
- The process should be composed of a quick design phase and should encourage the reuse of portions of existing design artifacts and applications in form of patterns; it should enable the automatic production of a consistent documentation at different levels of abstraction by re-engineering the produced code.
- The design aspects we decided to maintain from conventional PASSI are related to the initial part of the process (use case based requirements analysis) and the agent

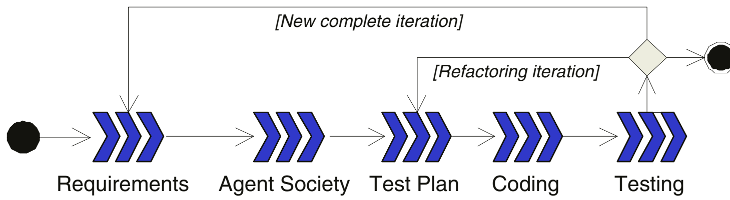


Fig. 3. The phases of the Agile PASSI process.

society model (functionality-based agents identification and a detailed domain ontology design). This satisfies the expectancy of already skilled PASSI designers that do not want to study a totally new process.

- Finally, the process has to be supported by a specifically conceived design tool in order to limit all the operations that are performed 'by hand' (this also includes design documentation production) because they contribute in significantly slowing down the process and could introduce mistakes in the final result.

The resulting process is reported in Figure 3 and it is composed of five different phases: (i) *Requirements* where the new iteration is planned (in terms of risks and requirements to be faced) and a use case based analysis of system requisites is performed; (ii) *Agent Society* where the agents that will constitute the system are identified and the domain application ontology defined; (iii) *Test Plan* where starting from requirements, a detailed plan of the test that will be applied to the code is prepared; (iv) *Coding* where code is produced (with patterns reuse); and (v) *Testing* where the produced portion of the system is tested accordingly to the previously prepared test plan.

4 PASSI Description

PASSI [9] is a process for multi agent systems development that covers all the design activities from the requirements analysis to the system implementation and deployment. The design work is carried out adopting five phases composed by twelve sequential and iterative work definitions used to produce the MAS specification.

The phases and work definitions of PASSI (in Figure 4 a SPEM diagram representing them) are:

1. **System Requirements**. It is composed of four different work definitions and produces a description of the functionalities required for the system and an initial decomposition of them accordingly to the agent paradigm. The four work definitions are: (i) the *Domain (Requirements) Description*, where the system is described in terms of functionalities; (ii) the *Agent Identification* where agents are introduced and the already identified requirements assigned to them; (iii) the *Role Identification* where agents' interactions are described using traditional scenarios; (iv) the *Task Specification* where the operational plan of each agent is draft.
2. **Agent Society**. It composes a model of the social interactions and dependencies among the agents of the solution. It is composed of four work definitions: in the *Domain Ontology Description* the elements occurring in the system domain are represented in terms of concepts, predicates, actions and relationships among them; in

the *Communication Ontology Description* the focus is on agent's communications that are explained in terms of referred ontology, content language and agent interaction protocol; in the *Role Description* distinct roles played by agents in the society and the involved tasks/behaviors are detailed; in the *Protocol Definition* non-standard agent interaction protocols are defined.
3. **Agent Implementation**. It is a model of the solution architecture in terms of required classes and methods. It is composed of four work definitions organized in two streams of activities (structure definition and behavior description) both performed at the single-agent and multi-agent levels of abstraction.
4. **Code**. It is a model of the solution at the code level. It is largely supported by patterns reuse and automatic code generation.
5. **Deployment**. It is a model of the distribution of the parts of the system across hardware processing units. The *Deployment Configuration* work definition, describes the allocation of agents in the units and any constraint on migration and mobility.

Testing in PASSI is divided in two different stages: the *Agent Test* where each single agent is tested after its implementation (Code phase) and the *Society Test* where the whole multi-agent system is tested (after the Deployment phase).

This great number of steps may take a long time to obtain the first prototype code. Also, the process is iterative both among the phases and in the whole life cycle; this configures PASSI as a traditional process in which the coding phase is positioned somehow late in the process and like many other classical approaches it is oriented to high level documentation production, and it is more adequate for projects with a low level of changes in requirements.

From PASSI we extracted several fragments some of which will be reused or adapted for the creation of the Agile PASSI process. In the following subsection we will describe the PASSI method fragments extraction process.

4.1 PASSI Fragments

Before performing the fragments extraction from PASSI, we re-engineered it in order to represent all the process aspects (activities, artifacts, constraints and conditions) in a way that could enable the method fragments identification. SPEM (Software Process Engineering Metamodel [6]) was adopted as a process meta-modeling language; this language allows an intuitive description of the software development process and its components and includes an UML profile that can be used to graphically represent the process using UML activity, class and use case diagrams. The core of SPEM is in its conceptual model: a software development process can be seen as a collaboration between abstract active entities called *Process Roles* that perform some operations called *Activities* on concrete entities called *Work Products*.

We represented the PASSI process in SPEM using two sequential steps, in the first we considered the whole process with the involved disciplines, in the second we detailed the separate phases and work definitions, following the conceptual model described above and the method fragment structure already defined in section 3. Starting from the procedural representation of PASSI composed of five phases (Figure 4), we decided to extract one different fragment for each one of the PASSI work definitions (refer

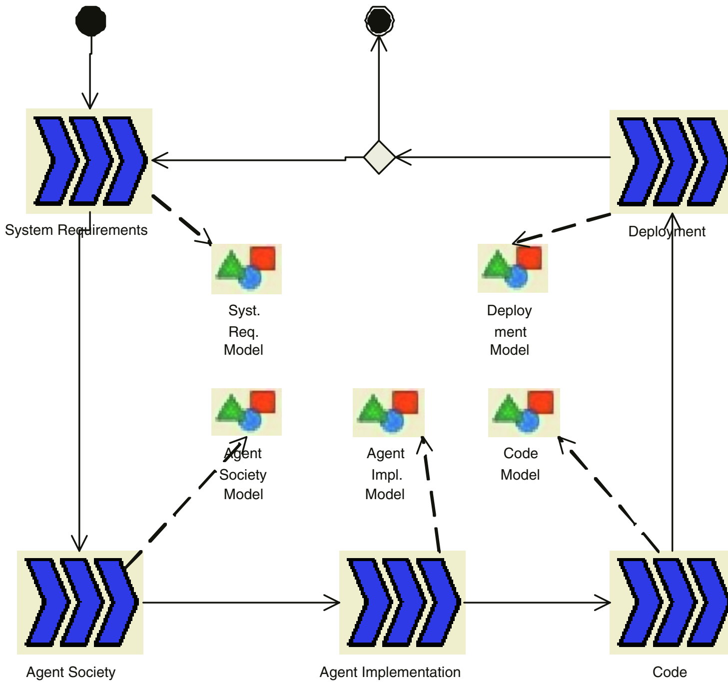


Fig. 4. The phases of the PASSI process.

to the beginning of this section for their list). In so doing we obtained a substantial simplification of our new process creation work: the assembly process will only deal with two levels of fragments (phase and work definitions); another consequence is that their modifications during fragments integration will be easier since it will mainly deal with work definition level fragments (and rarely with their composing activities).

At the end of our PASSI re-engineering and fragments extraction work we obtained seventeen work definition level fragments and five phase level ones; they constitute the fragments repository from which we selected the elements to compose the Agile PASSI process

5 The Resulting Agile PASSI Process

Starting for the considerations proposed in the previous sections we selected from the PASSI process some fragments that we consider in line with the new process requirements (see subsection 3.1) and our 'philosophy' in agents development (use case based agents identification and central role of ontology); the selected method fragments: Domain Requirements Description (a description of the system requirements in terms of use cases), Agent Identification (the clustering of system functionalities into packages associated to agents), Domain Ontology Description (an ontological description of the solution domain in terms of concepts, predicates and actions), Code reuse (a comprehensive pattern reuse technique that allows the automatic production of code) and Test-

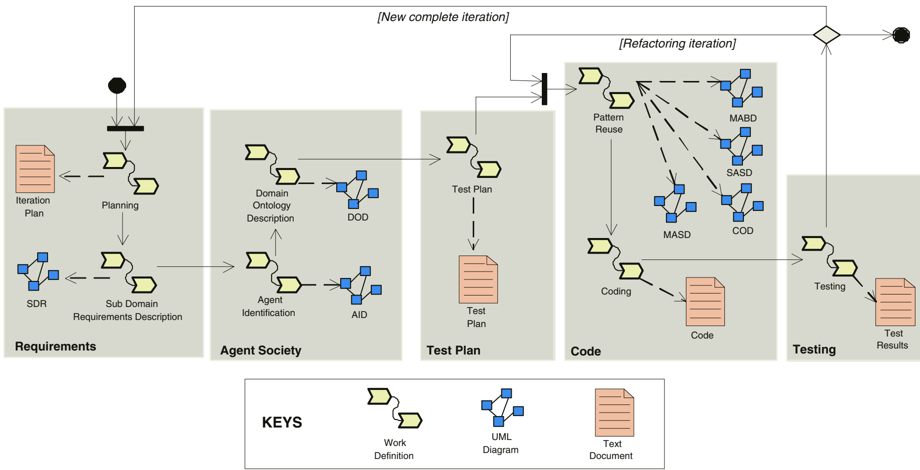


Fig. 5. The Agile PASSI process.

ing (of agents and societies). The new Agile Process (reported in Figure 5 in form of a SPEM activity diagram), resulting from the composition of these work definitions in the five phases of the general model proposed in Figure 3, is composed of eight work definitions (*Planning, Sub-Domain Requirements Description, Domain Ontology Description, Agent Identification, Pattern Reuse, Coding, Test Plan, Test*) and eleven artifacts (seven UML diagrams and four text documents).

More in details, the first phase (**Requirements**) consists in an high level analysis of the system under construction through two sequential work definitions:

– *Planning*, where through the communication among team elements and sequential iterations the problem is divided into sub-problems so to make possible a correct risks management and activities scheduling. This first activity should result in a text document (*Iteration Plan*) summarizing the considerations and the solution proposed by team elements.
– *Sub Domain Requirements Description*, a functional description of the system through common UML use case diagrams. This work definition corresponds to the PASSI *Domain requirements Description*, the 'Sub Domain' prefix has been added to stress the incremental concepts that are behind this process.

In the **Agent Society** phase, developer identifies the agents involved in the solution (assigning the previously identified functionalities to them), and then he defines the ontology of the domain. The phase is composed of two parts:

– *Agent Identification*, in this activity another use case diagram is composed, starting from the previously produced one, clustering use cases in packages that represent the functionalities assigned to agents; in this way, each agent will be responsible for the satisfaction of some requisites.
– *Domain Ontology Description*, the domain is expressed in terms of its ontology through a class diagram where classes represent concepts, predicates and actions.

We expect that this two work definitions are iteratively carried on ; after the identification of an agent, the definition of its knowledge and actions starts and this could bring to some changes in the list of functionalities assigned to it.

Testing is a continuous activity during an agile development process, in Agile PASSI it is divided in two phases; the first is **Test Plan**, that has been conceived referring to the agile processes principles and particularly to eXtreme Programming [19] rules; according to these rules the testing phase starts before the coding activity, the designer/programmer has to first prepare the test plans and then coding the component that must satisfy them (this will be proved during the following *Testing* phase).

The **Code** phase is composed of two strictly coupled parts.

– *Patterns reuse*, where we try to reuse portions of precedent projects through the reuse of patterns of services (interactions among agents), agents, tasks and actions. In this activity the Agent Factory tool proves very useful allowing us the automatic generation of relevant portions of code and a reduction of development time and costs.
– *Coding*, consists in the introduction of the code that cannot be derived from patterns (for instance problem specific algorithms).

Coding phase is the core of Agile PASSI and it is largely supported by a tool, APTK (Agile PASSI Toolkit), that is an add-in of a commercial design tool (Metaedit+). APTK offers several features to the designer, its main functionalities are:

– Automatic compilation of diagrams - this allows the partial drawing of some diagrams, for instance the Agent Identification diagram is initially drawn reporting the use cases of the previous work definition, and the complete design of some others starting from the code re-engineering and other design information (like applied patterns), for instance the Communication Ontology diagram is composed in this way.
– Support of changes - our tool, interacting with the Metaedit+ functionalities, allows the user to modify all the design models (even those automatically generated by the tool), and to profitably perform an incremental and iterative development of the project.
– Consistency check - the developer can perform a check on all the generated models to verify their consistency or he can use the MetaEdit+ checking feature for verifying the correctness and consistency of each single diagram.
– Report and project documentation generation - APTK allows the creation of MS Word or HTML documents representing all the design aspects.
– Patterns reuse - the user interacts with Agent Factory, that is totally integrated with APTK, to apply patterns to the system and generate their code.
– Automatic code generation and reverse engineering - Code generation and reverse engineering are entirely done by the Agent Factory application, through its integration in APTK.

Testing, after code completion, is the phase where the real test accordingly to the previously defined test plans is performed.

Agile PASSI has been created starting from conventional PASSI with the precise aim of having a lighter design process that could fit the needs arising from the development of small-medium size projects. As a consequence there are not fundamental differences between the two processes with the exception of those that we can individuate between a classic SEP and an agile one. Even one of the most agent-oriented aspects of a design process (the MAS meta-model) is not very different. In building Agile PASSI we referred to the MAS meta-model represented in Figure 2 whose elements are a subset of the conventional PASSI MAS meta-model [26].

Being our process agile, it is iterative, composed by a low number of steps and it strongly involves the end-user (or customer) during the development phases. These are choices we did in order to be compliant with the agile manifest principles[18], and as a consequence some of the phases of traditional SEPs are not considered (this is the case of the system architecture design that is left to the agent society organization) or performed very quickly. Quality assurance is enhanced by the large reuse of patterns, the automatic production of relevant portions of code and the consistency check performed by the tool on the design artifacts.

6 Conclusions and Future Works

This work started from the need of developing a new software engineering process (SEP) that could allow the quick prototyping of agent-oriented applications. In previous experiences we used the PASSI process that proved good for the development of medium-large size applications but it was too time consuming for the development of smaller size applications. This gave us the opportunity of applying our studies on the assembling of a new SEP by reusing parts (called method fragments) from other processes. This approach, already known as method engineering in the object-oriented context, is now diffusing in the agent-oriented community as a logical attempt of rationalizing and reusing the great number of development processes proposed in literature.

In this paper we discuss our approach that is composed of four phases: *Requirements Analysis, Process Model Design, Fragments Selection, Fragments Integration*. In these phases we also consider specific agency peculiarities like the MAS meta-model that differently from what happens in the object-oriented context is not a-priori known and fixed, but it is one of the most important differences that can be found in the development processes proposed in literature. The result of this work (the Agile PASSI process) is finally presented starting from the requirements that motivated its structure.

In the future we aim at further detailing the different aspects of our work, by formalizing a sufficient number of techniques and guidelines that could efficiently support the method engineer. As regards the Agile PASSI process, after having applied it in a couple of small projects, we can say that it fully achieved the goals we were pursuing from the methodological point of view, while the design tool (APTK) has still to be significantly improved in order to reach the flexibility and extensive support offered by the conventional PASSI support tool (PTK).

References

1. Brinkkemper, S., Lyytinen, K., Welke, R.: Method engineering: Principles of method construction and tool support. International Federational for Information Processing 65 **65** (1996) 336
2. Ralyte, J., Rolland, C.: An approach for method reengineering. Lecture Notes in Computer Science (2001) 27–30
3. Brinkkemper, S., Saeki, M., Harmsen, F.: Meta-modelling based assembly techniques for situational method engineering. Information Systems **24** (1999)
4. Board, I.S.: Ieee std 1074-1997, standard for developing software life cycle processes (1997)
5. Gilb, T.: Principles of Software Engineering Management. Addison-Wesley Reading (1988)
6. OMG: Software Process Engineering Metamodel Specification. http://www.omg.org (2002)
7. Cossentino, M., Sabatucci, L., Seidita, V.: Spem description of the passi process. Technical Report 20-03, ICAR-CNR (2003) Available online at http://www.pa.icar.cnr.it/ cossentino/ FIPAmeth/metamodel.htm.
8. Foundation for Intelligent Physical Agents: Method Fragment Definition. (2003)
9. Cossentino, M., Sabatucci, L.: Agent system implementation. In Paolucci, M., Sacile, R., eds.: Agent-Based Manufacturing and Control Systems: New Agile Manufacturing Solutions for Achieving Peak Performance, CRC Press (2004)
10. Cossentino, M., Sabatucci, L., Chella, A.: A possible approach to the development of robotic multi-agent systems. In: IEEE/WIC IAT'03 Conference, Halifax - Canada (2003)
11. M.Cossentino, L.Sabatucci, S.Sorace, A.Chella: Pattern reuse in the passi methodology. In: ESAW'03, Imperial College London, UK (EU) (2003)
12. Brinkkemper, S.: Method engineering: engineering the information systems development methods and tools. Information and Software Technology **37** (1995)
13. Kumar, K., Welke, R.: Methodology engineering: a proposal for situation-specific methodology construction. Challenges and Strategies for Research in Systems Development (1992) 257–269
14. Saeki, M.: Software specification & design methods and method engineering. International Journal of Software Engineering and Knowledge Engineering (1994)
15. Tolvanen, J.P.: Incremental method engineering with modeling tools: Theoretical principles and empirical evidence (ph.d. thesis). Jyväskylä Studies in Computer Science (1998) 301
16. Henderson-Sellers, B., Debenham, J.: Towards open methodological support for agent-oriented systems development. In Far, B., Rochefort, S., Moussavi, M., eds.: Proceedings of the First International Conference on Agent-Based Technologies and Systems, University of Calgary, Canada (2003) 14–24
17. Juan, T., Sterling, L., Winikoff, M.: Assembling agent oriented software engineering methodologies from features. In: Third International Workshop on Agent-Oriented Software Engineering, Bologna - Italy (2002)
18. Beck, K., al.M. Beedle, van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R., Mellor, S., Schwaber, K., Sutherland, J., Thomas, D.: (Agile manifesto) http://www.agilemanifesto.org.
19. Wells, D.: (Extreme programming - a gentle introduction) http://www. extremeprogramming.org.
20. O'Brien, P., Nicol, R.: Fipa - towards a standard for software agents. BT Technology Journal **16** (1998) 51–59
21. Paulk, M., Weber, C., Curtis, B.: The Capability Maturity Model for Software. Addison Wesley (1995)
22. Ghezzi, C., Jazayeri, M., Mandrioli, D.: Fundamentals of Software Engineering. Prentice Hall International (1991)

23. Boehm, B.: A spiral model of software development and enhancement. IEEE Computer **21** (1988) 61–72
24. Board, I.S.: Software life cycle processes (1998)
25. Ralyte, J., Rolland, C.: An assembly process model for method engineering. In: Proceedings of the 13th Conference on Advanced Information Systems Engineering, CAISE'01, Interlaken (Switzerland) (2001)
26. Bernon, C., Cossentino, M., Gleizes, M., Turci, P., Zambonelli, F.: A study of some multi-agent meta-models. In: Agent-Oriented Software Engineering Workshop (AOSE'04), New York (USA) (2004)

A Generative Approach for Multi-agent System Development

Uirá Kulesza[1], Alessandro Garcia[1], Carlos Lucena[1], and Paulo Alencar[2]

[1] PUC-Rio, Computer Science Department, LES, SoC+Agents Group,
Rua Marques de São Vicente, 225 - 22453-900, Rio de Janeiro, RJ, Brazil
{uira,afgarcia,lucena}@inf.puc-rio.br
[2] University of Waterloo, Computer Science Department, Computer System Group
Waterloo, Ontario, N2L 3G1 Canada
palencar@csg.uwaterloo.ca

Abstract. The development of Multi-Agent Systems (MASs) involves special concerns, such as interaction, adaptation, autonomy, among others. Many of these concerns are overlapping, crosscut each other and the agent's basic functionality. Over the last few years, several methodologies and implementation frameworks have been proposed to support agent-oriented software engineering. Although these approaches have brought some benefits to improve the productivity and quality on the MAS development, they present some restrictions. First, agent-oriented methodologies are too high level and do not indicate how to master the complexity of MAS concerns based on the object-oriented abstractions. Second, implementation frameworks provide object-oriented APIs for MAS development without providing guidelines for the modularization of agent concerns. Moreover, neither of the proposed agent oriented-approaches deals with the modeling and implementation of agent crosscutting concerns. This paper presents a generative approach for the development of MASs that addresses these restrictions. The proposed approach explores the MAS domain to enable the code generation of heterogeneous agent architectures. Aspect-oriented techniques are used to allow the modeling of crosscutting agent features. The generative approach brings several benefits to the code generation and modeling of agent crosscutting features since early development stages.

1 Introduction

Multi-Agent Systems (MASs) are composed of software entities that involve special properties (concerns), such as interaction, adaptation, autonomy, among others. With the growth of the Internet and the advances in networking technologies, the design and development of MASs have risen in importance. However, the effort and cost of designing and implementing MASs while satisfying quality requirements, such as maintainability and reusability, are still deep challenges to software engineers. [10, 14]. None is more serious than the difficulty to deal with the modularization and composition of multiple agent properties since early development stages [9, 13]. In general, a MAS has multiple software agents with different properties to be composed in different ways [9, 13]. Moreover these properties crosscut each other and the basic agent functionality, making their modeling, modularization, and composition more difficult.

Over the last few years, several methodologies and implementation frameworks for agent-oriented software engineering have been proposed. Agent-oriented methodolo-

R. Choren et al. (Eds.): SELMAS 2004, LNCS 3390, pp. 52–69, 2005.
© Springer-Verlag Berlin Heidelberg 2005

gies [15, 27] propose modeling languages and techniques that allow MAS developers to specify their systems using agent-oriented abstractions. Implementation frameworks [1, 18] improve the productivity of MAS development by providing customized object-oriented APIs. Despite the advantages that these agent-oriented software engineering approaches bring, each of them offers restrictions, such as: (i) most methods and modeling languages proposed are too high level and do not indicate how to master the complexity of these concerns based on the object-oriented abstractions; (ii) on the other hand, implementation frameworks do not provide guidelines for the modularization of agent concerns; (iii) agent-oriented methodologies and implementation frameworks also impose the use of MAS abstractions in the modeling and implementation of these systems considering a particular point of view; and (iv) moreover, neither of the proposed agent oriented-approaches deal with the modeling and implementation of agent crosscutting concerns typically encountered in MASs [9, 11-14]. As a consequence, these restrictions decelerate the development process and affect negatively the reuse and maintenance of the MAS artifacts.

In this context, we have explored the integrated use of two software engineering approaches for dealing with the mentioned restrictions: generative programming and aspect-oriented software development. We believe that the combination of these software engineering approaches can help to define a systematic and flexible MAS approach in order to overcome many of the restrictions presented by current MAS approaches.

Generative Programming (GP) [7] has been proposed recently as an approach based on domain engineering [17, 23]. It addresses the study and definition of methods and tools to enable the automatic production of software families from high-level specifications. GP promotes the separation of problem and solution spaces, giving flexibility to evolve both independently. Problem space models the existing concepts and features in a specific domain. Solution space consists of the components that are used to build particular software systems. Code generators represent the configuration knowledge in a generative model. They define how specific feature combinations in the problem space are mapped to a set of software components in the solution space.

The use of GP in the definition of a MAS approach offers the potential to explore the MAS domain systematically. It also allows the problem and solution spaces to evolve independently, including issues related to the technologies used. Therefore, it offers more flexibility to define and evolve the MAS high-level abstractions and architectures used in the production of particular system families for this domain. Moreover, GP advocates the clear definition of the mapping between high-level features and implementation components by means of code generators. In this way, GP addresses the lack of guidelines provided by agent-oriented methodologies during the translation of high-level agent features to specific combinations of implementation components. GP can also help to reduce the cost and effort of MAS development by means of the code generation of agent architectures.

Aspect-oriented software development (AOSD) [19, 26] is an evolving approach to modularize crosscutting concerns that existing paradigms (e.g.: object-oriented) are not able to capture explicitly. Crosscutting concerns are concerns that often crosscut several modules in a software system. AOSD encourages modular descriptions of complex software by providing support for cleanly separating the basic system func-

tionality from its crosscutting concerns. Aspect is the abstraction used to modularize the crosscutting concerns.

The use of aspect-oriented (AO) techniques makes it possible the modeling of several MAS concerns that are often scattered in the design and code of multi-agent systems and implementation frameworks [9, 11-14]. Thus, AO abstractions are used in our approach to capture several crosscutting concerns encountered in the implementation of agent architectures. Most of these crosscutting concerns are not captured by the existing agent-oriented methodologies. Examples of such concerns are interaction, autonomy, adaptation and collaborative roles. The use of AO techniques enhances the maintainability and reusability of MASs, because the design and code of crosscutting agent concerns can be modularized and are not intermingled with code of non-crosscutting agent concerns.

This paper presents an aspect-oriented generative approach for the development of MASs. Our generative approach explores specific features of the MASs domain to enable the code generation of agent architectures. It allows software engineers in dealing with several agent concerns in MASs from early development phases until the system implementation. The generative approach for MASs defines a domain-specific language, called Agent-DSL, for supporting the uniform modeling of several orthogonal (non-crosscutting) and crosscutting agent concerns. The approach also includes a generic and flexible aspect-oriented agent architecture [9, 13] to enable the generation of different kinds of agents. Aspect-oriented abstractions are used to design crosscutting concerns encountered in the architectural definition of an agent. Finally, the generative approach defines a code generator that maps abstractions of the Agent-DSL to specific compositions of objects and aspects in the agent architecture.

The remainder of this paper is organized as follows. Section 2 presents our generative approach for MASs, detailing the domain analysis and design phases of its development. Section 3 describes the implementation of the elements of the generative approach. Section 4 shows the application of the generative approach to a case study. Section 5 presents some related work. In Section 6, we offer our conclusions and indicate the direction of future work.

2 A Generative Approach for Multi-agent Systems

The aspect-oriented (AO) generative approach explores the domain of multi-agent systems (MASs) to improve their quality and productivity. The purpose of the generative approach is threefold: (i) to uniformly support crosscutting and orthogonal (non-crosscutting) features of software agents starting at early development stages; (ii) to abstract the common and variable features; and (iii) to enable the code generation of AO agent architectures.

Figure 1 depicts our generative approach that is composed of:

(i) a *domain-specific language* (DSL), called Agent-DSL, used to collect and model orthogonal and crosscutting features of software agents;
(ii) an *AO architecture* [9, 13] that models a family of software agents. It is centered on the definition of aspectual components to modularize the crosscutting agent features;
(iii) a *code generator* that maps abstractions of the Agent-DSL to specific compositions of objects and aspects in agent architectures.

The development of the generative approach underwent a typical domain engineering process [7, 23], organized in three phases: domain analysis (Section 2.1), domain design (Section 2.2), and domain implementation (Section 3).

In the domain analysis, common agent concerns encountered in MASs were modeled using feature models [17]. This study was supported by our extensive work on the development of several multi-agent systems [9-14] and on a survey of different modeling languages, MAS architectures and platforms [25]. Section 2.1 details the agent features modeled during domain analysis.

Domain design consisted of the specification of a generic and flexible AO agent architecture [9, 13]. Each feature modeled in domain analysis was considered. Section 2.2 presents the specification of the AO agent architecture that makes it possible to represent crosscutting agent concerns at the architectural level.

In the domain implementation, each of the elements of the generative approach was accomplished. Agent-DSL was implemented using Eclipse Modeling Framework (EMF) models [3]. The AO agent architecture was implemented as an AO framework using AspectJ and Java programming languages. Finally, the code generator of the generative approach was accomplished as an Eclipse plug-in. Section 3 details the implementation of these elements.

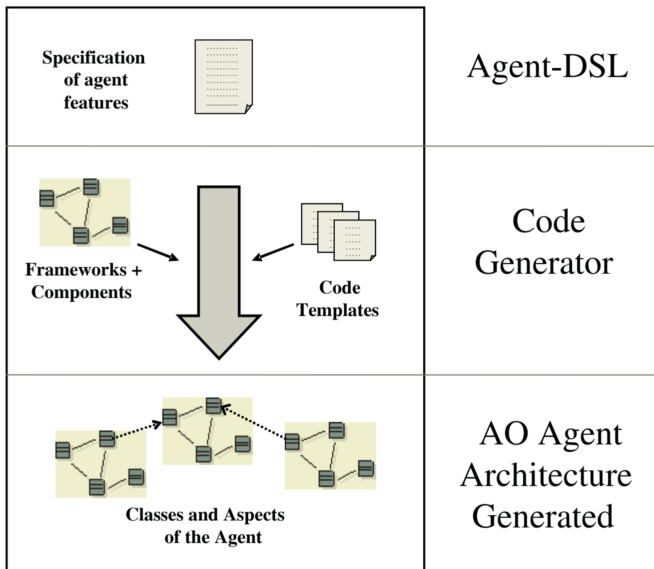


Fig. 1. The Generative Approach for MASs

2.1 Domain Analysis

During the domain analysis, recurring agent concerns of MASs were modeled using feature models [17]. Feature models are used to represent common and variable features of system families. We captured the different features associated with the agent concept, including non-crosscutting and crosscutting agent features. Figure 2 depicts a partial feature model produced during this phase.

A new kind of relation between features, called *crosscuts* relation, was introduced in feature models in order to support the representation of crosscutting features. We say that a feature A crosscuts a feature B, when either A or one of its subfeatures depends and inspects B or one of the subfeatures of B. The term “inspect” means the act of observing a feature. The following agent features were characterized as being crosscutting: interaction, adaptation, autonomy, and collaboration. Each of them inspects elements of the knowledge feature in order to exhibit a specific agent property. For example, the autonomy feature inspects changes on the knowledge feature in order to detect the need for autonomous proactive behavior. Figure 2 presents the crosscutting relationships between features and respective inspected features.

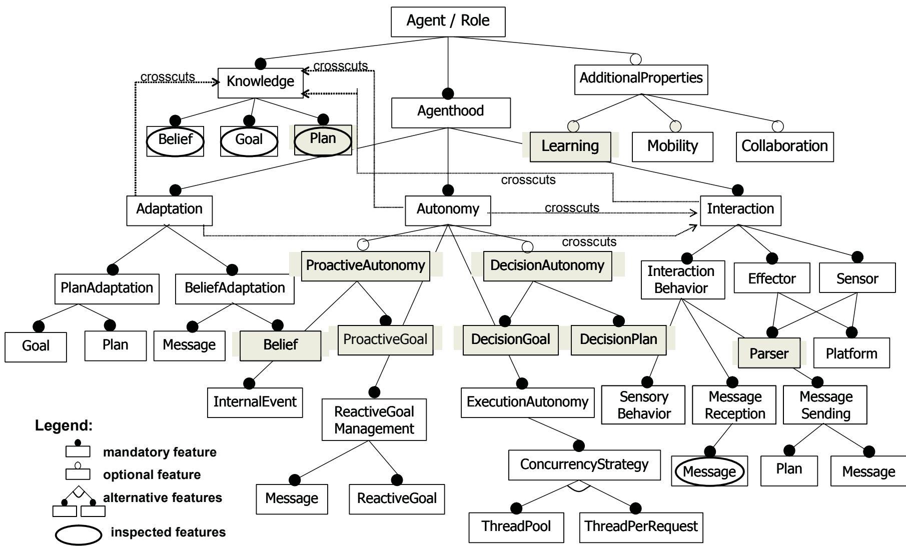


Fig. 2. Partial Feature Model of the Agent

Tables 1, 2 and 3 present the definition of the agent features presented in Figure 2. Table 1 emphasizes the knowledge features, Table 2 describes the concerns common to every kind of software agent (agenthood features), and Table 3 defines the optional agent features (additional properties). The tables also indicate which agent concerns are crosscutting and detail the respective inspected features. Learning and mobility are not mandatory features in agent architectures and they have not been explored in this work.

Table 1. Agent Knowledge Features

Agent Knowledge		
Feature	Description	Crosscutting?
Belief	• Describes information about the agent itself and about the external environment.	No
Goal	• Represents a specific agent aim to be achieved.	No
Plan	• Set of actions to achieve an agent goal. • The plan execution involves the manipulation of beliefs.	No

Table 2. Agenthood Features

Agenthood (Basic Properties)		
Feature	Description	Crosscutting?
Interaction	• Defines the agent ability to communicate with the environment. • The agent receives and sends messages to the environment by means of its sensors and effectors. • External messages are translated to the agent ontology using specific parsers. • Parsers translate internal messages to a specific external representation.	Yes • It crosscuts, for example, the Knowledge feature to inspect agent plans that need to send external messages.
Adaptation	• Defines changes in the agent knowledge or behavior. • It encompasses belief adaptation and plan adaptation. • *Belief adaptation* is responsible for interpreting received messages from the environment and for manipulating its beliefs based on the message contexts. • *Plan adaptation* determines the plan the agent must execute whenever a new goal needs to be achieved.	Yes • It crosscuts the Knowledge feature to inspect belief updates. • It also crosscuts the Interaction feature to inspect messages received.
Autonomy	• Instantiates and manages the agent goals. • It deals with three types of goals: reactive goals, proactive goals, and decision goals. • *Reactive goals* are instantiated when the agent receives an external request from other agents or environment components. • *Proactive goals* are instantiated due to internal events that occur, such as the end of a plan execution or the achievement of a specific agent state. • *Decision goals* are instantiated due to external or internal events and are used to decide whether special reactive or proactive goals could be instantiated.	Yes • It crosscuts the Knowledge feature to inspect belief updates. • It also crosscuts the Interaction feature to inspect received external messages.

Table 3. Agent Additional Properties Feature

Agent Additional Properties		
Feature	Description	Crosscutting?
Collaboration	• The agent ability to cooperate with other agents through the performance of roles. • A role introduces to the agent extra features of knowledge, interaction, adaptation and autonomy. • Each agent can play different roles.	Yes

2.2 Domain Design

Domain design consisted of specifying a generic and flexible agent architecture for the domain at hand. Our domain design considered all the features modeled during domain analysis. The AO agent architecture is a refinement of a previous work [11, 14]. It uses two kinds of components: (i) the Knowledge component that modularizes the orthogonal features associated with the agent knowledge; and (ii) the *aspectual components* that separate the crosscutting agent features from each other and from the Knowledge component. Aspectual components represent crosscutting features at the architectural level.

Figure 3 depicts the components of the AO agent architecture. We have used a new notation to graphically represent an AO architecture. It is an extension of the ASideML modeling language [4]. We developed this notation to enable the representation of aspectual components. An aspectual component may crosscut other aspectual or non-aspectual components using its crosscutting interfaces. A crosscutting interface specifies when and how the aspect affects one or more architectural components. A crosscutting interface may both add new state or behavior in other components and intercept (and modify) the behavior of components. Non-aspectual (normal) components are represented in a similar way to UML [2] and offer their services through the normal interfaces.

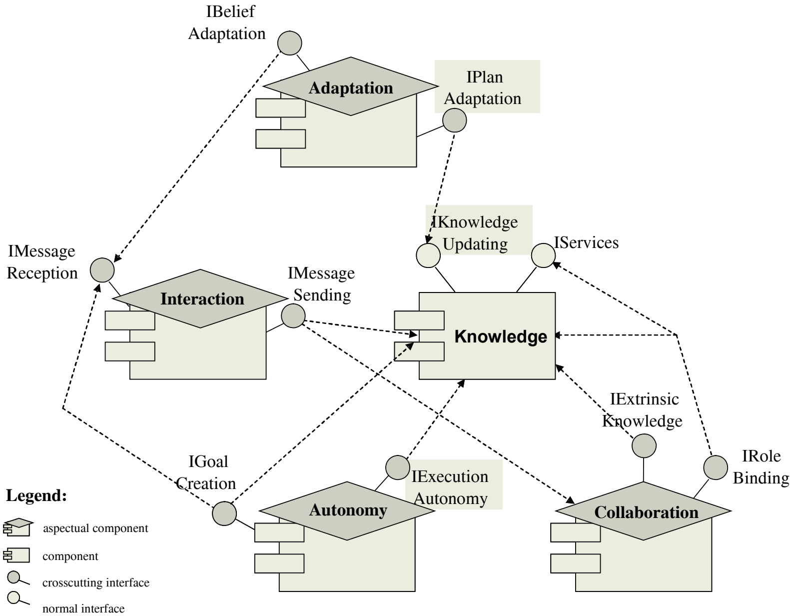


Fig. 3. The Aspect-Oriented Agent Architecture

The Knowledge component models the orthogonal features (belief, goal, plan) related to the knowledge feature. It realizes two normal interfaces: (i) `IKnowledgeUpdating` – to update the agent knowledge; and (ii) `IServices` – to offer agent services. The implementation (section 3.2) of this component is refined as a set of classes.

Each aspectual component was refined during the domain implementation (Section 3.2) as a set of aspects and auxiliary classes, which are also part of the crosscutting feature. The Interaction aspectual component models the interaction crosscutting feature. It is composed of two crosscutting interfaces: (i) `IMessageReception` – which introduces the capacity to receive external messages into the Knowledge component; and (ii) `IMessageSending` – which crosscuts elements of the Knowledge component to define specific points where it is necessary to send messages to the environment. It also crosscuts elements of the Collaboration aspectual component to specify specific points in collaboration plans where it also is necessary to send messages to the environment.

The Adaptation aspectual component models the adaptation crosscutting feature. It is composed of two crosscutting interfaces: (i) `IBeliefAdaptation` – which intercepts services of the `IMessageReception` interface on the Interaction component in order to update agent beliefs when new external messages are received; and (ii) `IPlanAdaptation` – which intercepts services of the `IKnowledgeUpdating` interface on the Knowledge component in order to instantiate new plans when the agent needs to achieve a specific goal.

Finally, the Collaboration aspectual component models the role crosscutting feature. It is composed of two crosscutting interfaces: (i) `IExtrinsicKnowledge` – which introduces new knowledge associated with the roles in the Knowledge component; and (ii) `IRoleBinding` – which defines specific points in the Knowledge component where agent roles are instantiated and bound to the agents.

3 Implementing the MAS Generative Approach

In this section, we describe the implementation of the elements of our generative approach: (i) the Agent-DSL; (ii) the aspect-oriented agent architecture; and (iii) the code generator.

3.1 Agent-DSL

Based on the feature models defined in the domain analysis (Section 3.1), we defined a domain-specific language, called Agent-DSL. This language is used to specify the agency properties that an agent could have to accomplish its tasks. It makes it possible to model agent features, such as knowledge, interaction, adaptation, autonomy and collaboration.

The Eclipse Modeling Framework (EMF) [3] was used to specify the Agent-DSL. EMF is a Java/XML framework for generating tools and other applications based on simple class models. By using this framework, it was necessary to define a model that expresses the semantics of the Agent-DSL - in other words, the meta-model of this DSL. EMF allows the specifying of a meta-model by using XML Schema, annotated Java or UML modeling tools (e.g.: Rational Rose). After that, EMF uses this meta-model specification to automatically generate Java code and Eclipse editors, which allows to create and edit models that conform to this meta-model.

In order to use EMF to specify the Agent-DSL, we first translated the feature models to a UML class diagram. In this translation, we basically converted: (i) features to classes; (ii) mandatory feature relations to UML composition relations; and (iii) optional feature relations to UML aggregation relations. Also, specific characteristics of each feature were introduced as class attributes. Then, this class diagram served as input for EMF generating: (i) Eclipse visual editors that permit the creation and manipulation of Agent-DSL models; and (ii) Java classes to manipulate the instances of Agent-DSL models. The latter are used by the code generator of the generative approach (see details in Section 3.3), during the customization of agent architectures.

Figure 4 presents the EMF visual editor that is used to create instances of Agent-DSL models. The figure illustrates the specification of an agent used in a case study. Section 4 presents additional details about this case study.

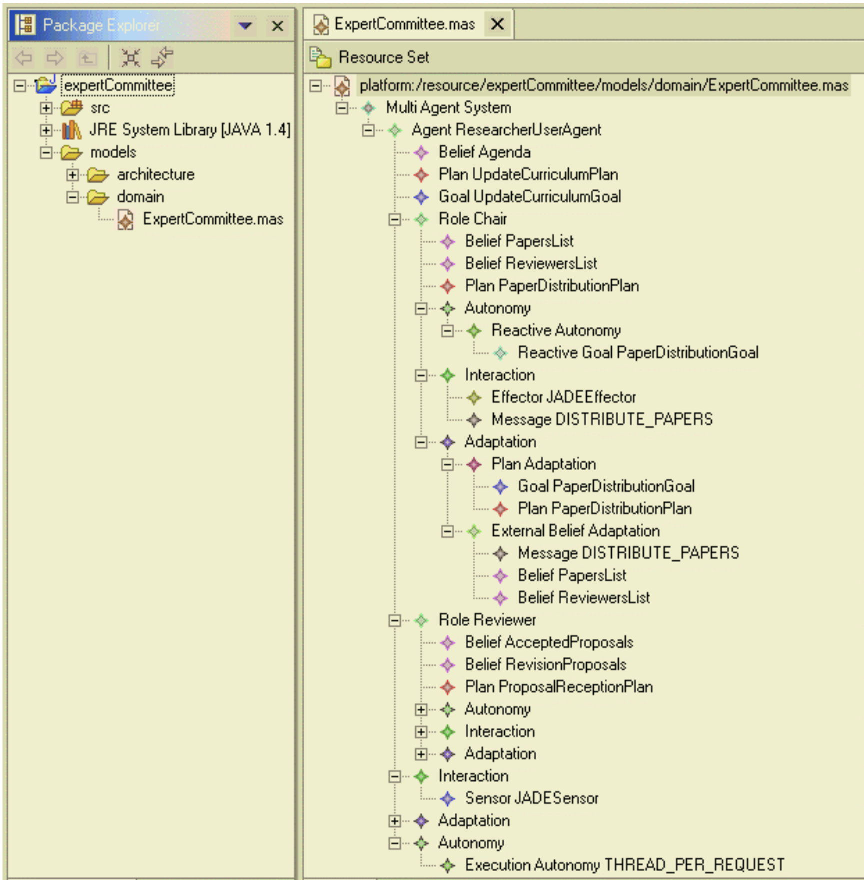


Fig. 4. An Agent Specification using Agent-DSL

3.2 The Aspect-Oriented Agent Framework

The implementation of the generic AO agent architecture (Section 2.2) was realized using Java and AspectJ [20] programming languages. The basis of the architecture implementation is an AO framework that contains hot-spots as classes and aspects [8]. Figure 5 presents a partial description of the AO framework. Every class and aspect presented in the figure is a hot-spot. The ASideML modeling language [4] is used to represent visually the framework. This language extends UML with notations for representing aspects.

The notations provide a detailed description of the aspect elements. In this modeling language, an aspect is represented by a diamond; it is composed of internal structure and crosscutting interfaces. The internal structure declares the internal attributes and methods. At the detailed design level, a crosscutting interface specifies when and how the aspect affects one or more classes [4]. Each crosscutting interface is composed of inter-type declarations, pointcuts and advices. The first part of a crosscutting interface represents inter-type declarations, and the second part represents pointcuts

and their attached advices. The notation uses a dashed arrow to represent the *crosscut* relationship, which relates one aspect to classes and/or aspects.

The Knowledge component (Section 2.2) was refined as a set of classes – `Agent`, `Belief`, `Goal` and `Plan` classes. Each of them represents a specific hot-spot that can be extended to define an agent type. Agent beliefs are defined in our architecture as domain classes that `Agent` instances can aggregate. Each one of the aspectual components (Section 2.2) was refined as a central aspect and a set of auxiliary classes. Figure 5 only presents the main aspects that refine the agent knowledge classes incorporating specific agent features.

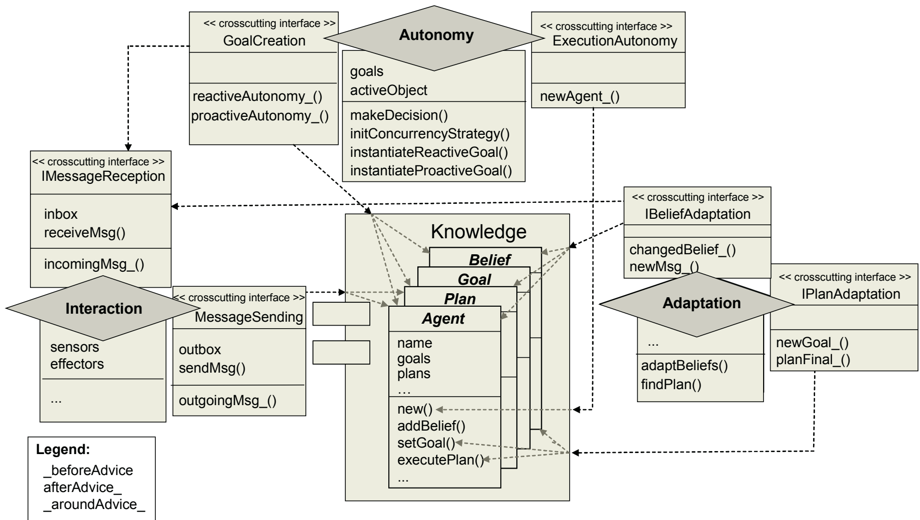


Fig. 5. The Aspect-Oriented Agent Framework

The Interaction component is defined as an abstract aspect that introduces interaction capabilities (inbox, outbox, sensors, effectors, parsers) in the `Agent` class. It also intercepts domain classes and sensors in the agent environment to enable the message reception by means of AspectJ pointcuts and advices. Finally, the `Interaction` aspect defines two abstract pointcuts and some abstract methods. The abstract pointcuts are used to define specific points in role aspects and plan classes where internal messages must be sent. The abstract methods are specialized to create and initialize specific sensors and effectors. The `Interaction` subaspects define the concrete configuration of the `Interaction` aspect by implementing the abstract pointcuts and methods. It is possible to specify a different `Interaction` subaspect for each one of the agent types or roles defined in an MAS.

The Adaptation component defines the `Adaptation` abstract aspect, which enables the `Agent` class to adapt its beliefs and plans. The belief adaptation of the `Adaptation` aspect is defined by intercepting the `receiveMsg()` method of the `Agent` class (introduced by the `Interaction` aspect). After that, specific advices and methods are responsible for updating beliefs based on external messages received by the agent. The plan adaptation, defined in the `Adaptation` aspect, intercepts the `setGoal()`

method of the `Agent` class and the erroneous execution of the `execute()` method of the `Plan` subclasses. The purpose is to determine new agent plans to be executed by the agent to reach a specific goal. The `Adaptation` abstract aspect also offers abstract methods to be defined by subaspects. These subaspects allow defining specific belief and plan adaptation for each one of the agent types or roles in MASs.

The Autonomy component defines the `Autonomy` aspect, which enables the `Agent` class to instantiate and manage reactive goals and to execute concurrently several plans (execution autonomy). However, for sophisticated agent types, the `Autonomy` aspect also allows defining proactive and decision autonomy. To instantiate reactive goals, the `Autonomy` aspect also intercepts the `receiveMsg()` method of the `Agent` class. This interception is used to verify if specific external events (for instance, another agent's request) demand the instantiation of reactive goals. The execution autonomy is implemented in the `Autonomy` aspect by defining an Active Object [16], which monitors the `Agent` class's list of plans to perform to execute them in separate threads. The proactive autonomy is implemented by specifying: (i) several pointcuts in agent knowledge classes that represent specific events of interest, and (ii) an advice associated with these pointcuts, which is responsible for determining if a proactive goal must be instantiated in the occurrence of any of these events. Finally, the decision autonomy only defines a `makeDecision()` method in the `Autonomy` aspect that is invoked in the advices associated with the pointcuts of reactive and proactive goal instantiation. This method verifies whether it is necessary to execute a decision plan upon the occurrence of a specific event or upon receipt of a message. `Autonomy` subaspects can also be implemented to define specialized proactive, reactive and decision autonomy for each one of the agent types and roles defined in a MAS.

The Collaboration component is implemented by defining role aspects that introduce attributes and methods in an agent type (`Agent` class or subclass). These elements respectively define specific beliefs and behaviors of roles. Also, specific `Plan` and `Goal` subclasses must be defined for the roles. The plans defined for a role manipulate the attributes (beliefs) and invoke methods (behaviors) introduced by the role aspect. `Goal` classes specified for a role are instantiated by an `Autonomy` subaspect that is specially created for the role. Specific `Interaction`, `Adaptation` and `Autonomy` subaspects can be defined for an agent role. Section 4 exemplifies the definition of subaspects for agent roles of a specific case study developed by our research group.

Besides the framework, some components were created to implement specific functionalities associated with the agenthood features, such as:

- interaction feature: concrete sensors and effectors specially tailored to specific agent platforms (such as JADE [1]);
- autonomy feature: concrete concurrency strategies (such as "thread pool" and "a thread per request") used by the active object [16] to implement the agent's execution autonomy.

3.3 The Code Generator

In the configuration knowledge of the generative approach, we implemented a code generator as an Eclipse plug-in [24]. This generator maps abstractions of Agent-DSL

models into the components of the AO agent architecture. The AO framework (Section 3.2) supports the implementation of agent architectures. The main task of the generator is to instantiate the framework for a given multi-agent system. It creates subclasses and subaspects for specific hot-spots of the framework. Depending on the Agent-DSL model provided, the code relative to new agent types (or roles) and their respective agent properties are generated.

The implementation of the code generator was accomplished by using EMF technology. The EMF representation of the agent architecture was supported by the definition of an architectural model. This model is responsible for specifying an architecture that will be generated. An architectural model aggregates the components of an architecture. Each component is composed of classes, aspects and templates. Templates make it possible to define some class or aspect that needs to be customized, based on information collected by a DSL. The architectural model was implemented as an EMF model, similar to the Agent-DSL (Section 3.1).

Figure 6 shows the architectural model of the AO agent framework. It is composed of the classes and aspects defined in the framework. This architectural model also contains several templates that are used to express structure and behavior of classes and aspects that we want to generate. Java Emitter Templates (JET) a generic template engine of the EMF, was used to implement and process the templates. Examples of templates are: (i) concrete instances of hot-spots (classes or aspects), such as specific agent type classes, specific agenthood subaspects; (ii) specific agent plans and goals classes; and (iii) specific role aspects. The Agent-DSL collects the information required in the code generation to customize these templates for each specific agent. Each code template defines the specific information of the Agent-DSL model to be used during its customization.

Thus the code generator uses two EMF models to instantiate the AO framework: an Agent-DSL model and an architectural model. Different agent architectures can be generated depending on the Agent-DSL and the architectural model informed by the agent developers. The plug-in of the code generator includes a wizard in the Eclipse workbench to start the process of code generation. The wizard requests from the user: (i) a source folder in a Java project to store the classes and aspects generated, (ii) the Agent-DSL model of the MAS to be generated, and (iii) the architectural model that describes the AO agent framework. During the generation process, the code generator traverses the architectural model and it proceeds as follows: (i) for each component encountered it generates a correspondent Java package; (ii) for each class and aspect encountered it loads the correspondent element in the Java project; (iii) for each template encountered it processes this element using the information collected by the Agent-DSL model, and it loads the final element generated (class or aspect) in the specified Java project. Although we have used the architectural model of the AO agent framework in the definition of our generative approach, a different architectural model could be used during the generation process. In this case, different classes, aspects and templates could be included in the architecture model. Also, the customization of these new templates could be redefined based on information collected by the Agent-DSL model.

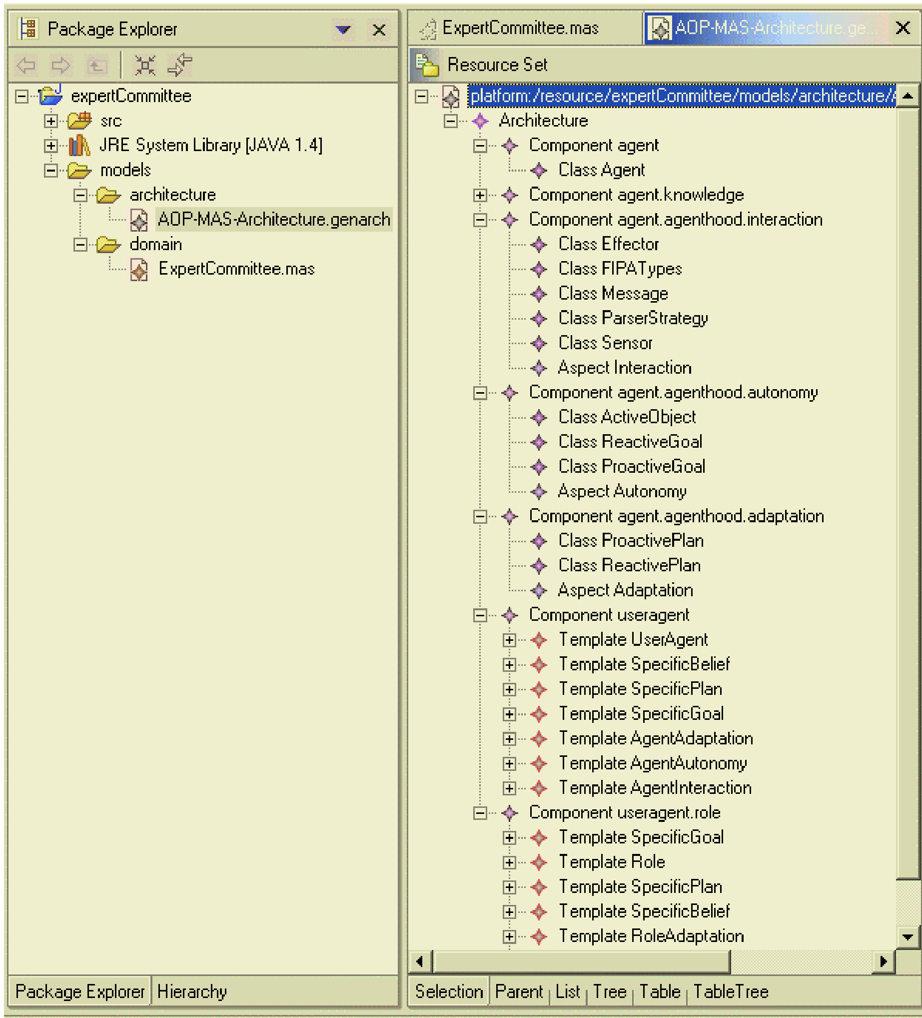


Fig. 6. The Architectural Model of the AO Agent Framework

4 Using the Generative Approach in a Case Study

We have used the generative approach for the development of the ExpertCommittee (EC) system, which is a case study undertaken by our research group [9]. EC is an open system that supports the management of paper submissions and the reviewing process for a conference. Software agents have been introduced to EC in order to assist its users with time-consuming activities and automate repetitive user tasks. EC agents are software assistants that represent paper authors, chairs, PC members and reviewers and coordinate their activities. The EC system also includes information agents. JADE [1] is used as the communication platform in this system. As a consequence, the EC system encompasses sensors and effectors for the JADE platform.

Figures 7 and 8 show several elements generated for the EC system. Several classes and aspects are generated based on its Agent-DSL model and on the JET source templates included in the architecture model of the AO agent framework (section 3.3). First, it is generated the `ResearchUserAgent` class, which represents a specific agent type. The roles played by instances of this class are also generated. The figures represent two of them: the `Chair` and `Reviewer` aspects. Each role aspect introduces the role-specific knowledge in the `ResearchUserAgent` class. Specific `Plan` and `Goal` classes are also generated to the two roles. Figure 7 depicts the specific agent, roles and plans generated for the EC system. The code templates used to generate each of these classes and aspects are customized using the specific Agent-DSL model for the EC system. The `ResearcherUserAgent` class, for example, is customized using the agent name and beliefs collected by the Agent-DSL model. On the other hand, the `Chair` and `Reviewer` aspects use the role name and role beliefs.

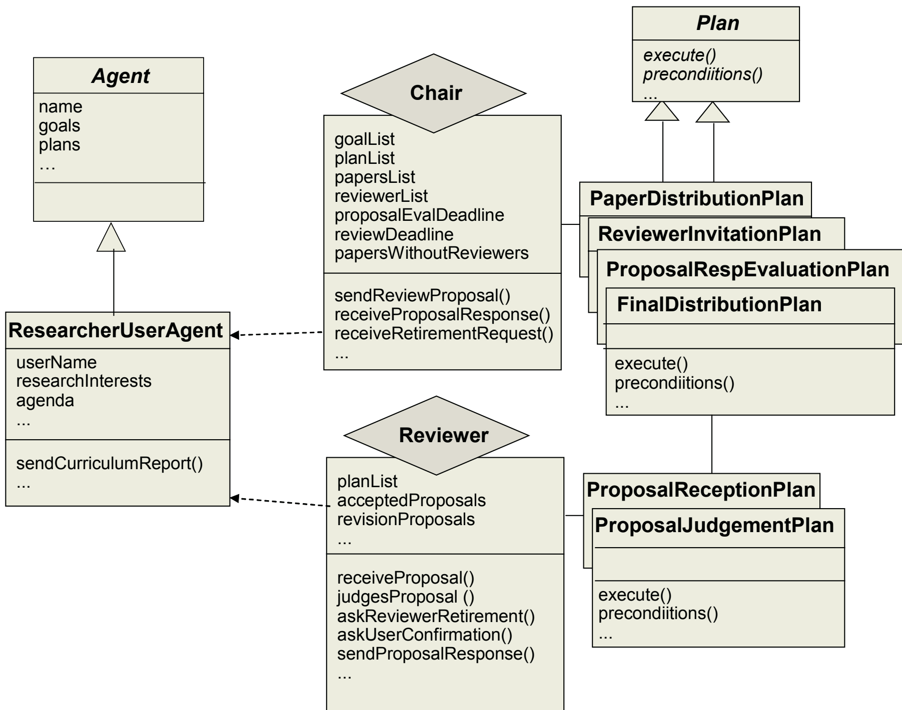


Fig. 7. Specific Agent and Roles Generated for the ExpertCommittee MAS

Different `Interaction`, `Adaptation` and `Autonomy` subaspects are generated for each one of the roles in the EC system. Figure 8 presents these subaspects. For instance, the `ChairInteraction`, `ChairAdaptation` and `ChairAutonomy` aspects are produced to agents playing the chair role. `ChairInteraction` initializes JADE sensors and effectors to be used by the agents playing the chair role. `ChairAdaptation` realizes specific belief and plan adaptation of the chair role. Finally, `ChairAutonomy` defines: (i) a reactive autonomy – to instantiate specific goals when receiv-

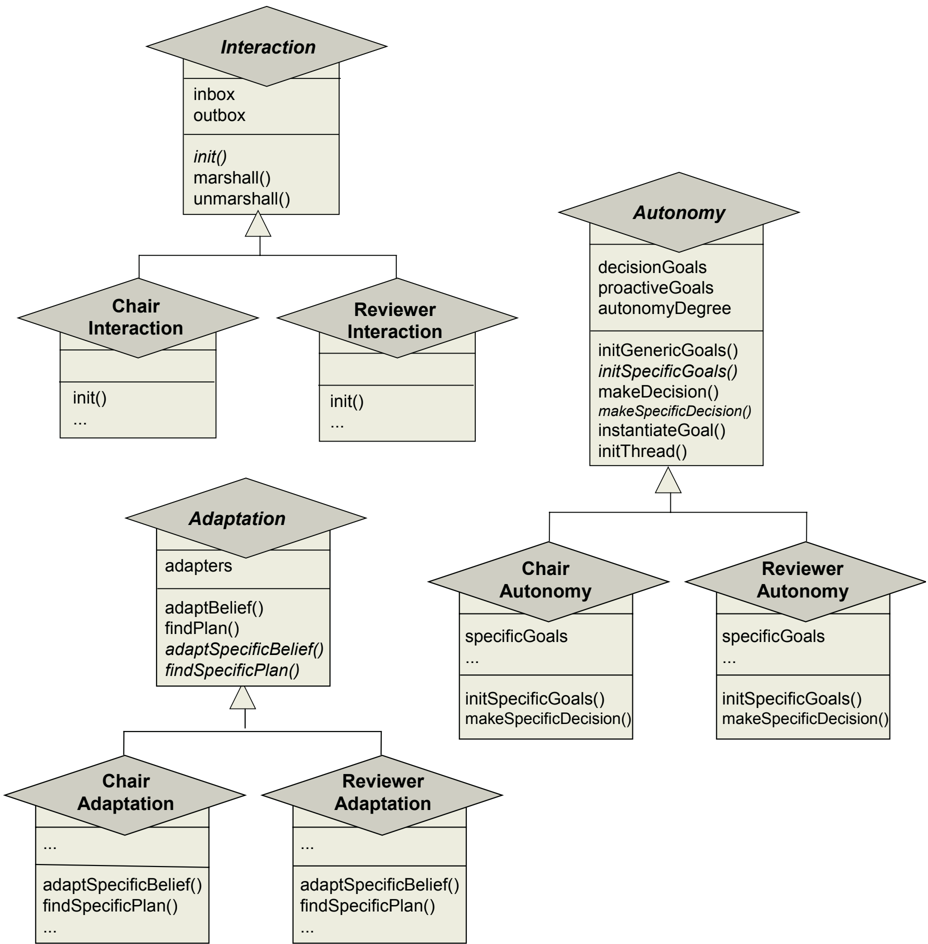


Fig. 8. Specific Agenthood Subaspects Generated for the ExpertCommitte MAS

ing external messages from reviewer agents; (ii) a proactive autonomy – to instantiate specific goals when internal events occur; and (iii) an execution autonomy – which defines a “thread per request” concurrency strategy to execute agent plans.

5 Related Work

Agent-based software engineering has been studied from different perspectives, including agent-oriented methodologies and languages for higher-level development phases [15, 27], and implementation frameworks [1, 18]. Existing agent-oriented methodologies, such as Gaia [28], provide limited support to deal with concerns that are internal to software agents. In addition, they usually do not address the code generation of MASs from high-level specification. Implementation frameworks provide object-oriented APIs for MAS development, without providing guidelines for the

modularization of agent concerns. Besides these frameworks do not deal with the modeling and implementation of agent crosscutting concerns typically encountered in MASs [9, 11-14].

Cossentino et al [6] have proposed PASSI (Process for Agent Societies Specification and Implementation), a methodology to specify, design and implement MASs. This methodology proposes the organization of the MAS development process in different phases from the requirements specification through to system deployment. Each phase focuses on the definition or refinement of a system model. Many PASSI models are adaptations of UML standard models, such as use-case, class and activity diagrams, which incorporate agent-oriented abstractions. The use of class diagrams in the design of MASs brings the facility to generate the skeletons of many classes of the system. The authors have also explored the reuse of recurring agent design patterns to improve the quantity and quality of code generated. However, the PASSI approach does not support the systematic modularization and generation of code relative to crosscutting agent concerns.

Pace et al [22] have developed the Smartweaver approach. Their approach provides assistance for the development of MAS applications by means of integration of agent-oriented and aspect-oriented frameworks. The authors demonstrate the application of their approach which consists of two components: (i) Bubble [22] – an agent-oriented framework used to the implementation of reactive agents; and (ii) Aspect-Moderator [5] – an AO framework that supports the coordination between functional components and aspects. Aspect-Moderator is used to capture typical crosscutting concerns, such as concurrency, logging, and event handling. The Smartweaver approach systematically addresses the incorporation of aspects in agent models. However, the code generation is limited and it does not support essential agent concerns, including autonomy, interaction, and adaptation.

6 Conclusions and Ongoing Work

We have presented a generative approach for the development of MASs. The main purpose of our approach is to explore the domain of MASs to enable the code generation of agent architectures. The generative approach makes it possible to deal with orthogonal (non-crosscutting) and crosscutting agent features since early development phases in a uniform way.

Compared with existing MASs development approaches [1, 15, 18, 27], our work brings important benefits. The generative approach is flexible in the sense that it allows the problem and solution spaces to evolve independently. In problem space, new DSLs can be created to address different agent features or current DSLs can be adapted to address a better understanding of a specific agent feature or to add specificities of an MAS. Moreover, the generative approach is also very practical. It defines clearly the mapping between high-level features and implementation components (classes, aspects) of agent architectures in the code generator. It also offers a clear separation of orthogonal and crosscutting agent features in problem and solution spaces.

The use of aspect-oriented technologies in the agent architecture also brings valuable advantages to our approach. The implementation of the code generator was sim-

plified because crosscutting agent features in the Agent-DSL are directly mapped to aspect-oriented abstractions. Using only object-oriented abstractions, crosscutting agent features need to be composed in the code of classes by the code generator. It makes the generator more complex and difficult to be implemented.

The work presented in this paper represents the current status of the generative approach. Different studies are being developed to improve our capacity to generate MAS architectures. In the problem space, we are extending the Agent-DSL to allow for the modeling of other relevant MASs concerns, such as, agent coordination, learning and mobility. In the solution space, an ongoing research project is the definition of an architectural definition language (ADL) that supports the definition of modular and aspectual components. This ADL will be used to formalize aspect-oriented architectures. We claim to enable the specification of AO architectures at a high-level independent of technologies, similar to MDA [21].

Regarding the configuration knowledge (code generators) of the generative approach, we intend to offer flexible ways to specify the transformations of: (i) elements in the Agent-DSL to elements (components, aspects and interfaces) of the ADL; and (ii) elements of the ADL to concrete technologies (for instance, Java and AspectJ). Finally, we plan to develop new and more complex case studies in order to better evaluate the usability and usefulness of our generative approach.

Acknowledgements

This work has been partially supported by CNPq under grant No. 140252/2003-7 for Uirá, grants No. 141457/2000-7 and No. 381724/2004-2 for Alessandro, and by FAPERJ under grant No. E-26/150.699/2002 for Alessandro. The authors are also supported by the PRONEX Project under grant 7697102900, and by ESSMA under grant 552068/2002-0 and by the art. 1 of Decree number 3.800, of 04.20.2001. This research has also been partially supported by the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

1. F. Bellifemine, A. Poggi, G. Rimassi. “JADE: A FIPA-Compliant Agent Framework.” Proc. Practical Applications of Intelligent Agents and Multi-Agents, pp. 97-108, April 1999.
2. G. Booch, I. Jacobson, J. Rumbaugh. “Unified Modeling Language - User's ad Guide”. Addison-Wesley,1999.
3. F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, T. Grose. “Eclipse Modeling Framework”. Addison-Wesley, 2003.
4. C. Chavez. “A Model-Driven Approach to Aspect-Oriented Design”. PhD Thesis, PUC-Rio, April 2004.
5. C. Constantinides, A. Bader, T. Elrad, M. Fayad. “Designing an Aspect-Oriented Framework”. Computing Surveys, 32:41, 2000.
6. M. Cossentino, M. Potts. “A CASE tool supported methodology for the design of multi-agent systems.” In Proc. of the 2002 International Conference on Software Engineering Research and Practice (SERP'02), Las Vegas, USA, June 2002.
7. K. Czarnecki, U. Eisenecker. Generative Programming: Methods, Tools, and Applications, Addison-Wesley, 2000.

8. M. Fayad, D. Schmidt, R. Johnson. “Building Application Frameworks: Object-Oriented Foundations of Framework Design”. John Wiley & Sons, September 1999.
9. A. Garcia. “From Objects to Agents: An Aspect-Oriented Approach.” PhD Thesis, PUC-Rio, April 2004.
10. A. Garcia, C. Lucena. “Software Engineering for Large-Scale Multi-Agent Systems”. ACM Software Engineering Notes, August 2002.
11. A. Garcia, et al. “Engineering Multi-Agent Systems with Aspects and Patterns.” Journal of the Brazilian Computer Society, September 2002.
12. A. Garcia et al. “Separation of Concerns in Multi-Agent Systems: An Empirical Study.” In: C. Lucena et al (Eds), “Advances in Software Engineering for Multi-Agent Systems.” Springer-Verlag, LNCS 2940.
13. A. Garcia, U. Kulesza, C. Lucena. “Separation of Concerns in Open Multi-Agent Systems: An Architectural Approach.” Proceedings of the SELMAS’04, Edinburgh, May 2004.
14. A. Garcia, C. Lucena, D. Cowan. “Agents in Object-Oriented Software Engineering.” Software: Practice and Experience, May 2004, pp. 1-33.
15. C. Iglesias, et al. “A Survey of Agent-Oriented Methodologies.” Proceedings of the ATAL-98, Paris, France, July 1998, pp. 317-330.
16. R. Lavender, D. Schmidt. “Active Object: an Object Behavioral Pattern for Concurrent Programming.” In: Pattern Languages of Program Design, Addison-Wesley, 1996.
17. K. Kang, S. Cohen, J. Hess, W. Novak, A. Peterson. “Feature-Oriented Domain Analysis (FODA) Feasibility Study.” Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.
18. E. Kendall, et al. “A Framework for Agent Systems.” Implementing Application Frameworks – Object-Oriented Frameworks at Work, M. Fayad et al. (eds.). John Wiley & Sons: 1999.
19. G. Kiczales, et al. “Aspect-Oriented Programming.” Proc. Of`ECOOP’97, LNCS 1241, Springer-Verlag, Finland, June 1997.
20. G. Kiczales, et al. “Getting Started with AspectJ.” Communication of the ACM. October 2001.
21. J. Miller, and J. Mukerfi, MDA Guide Version 1.0, Object Management Group, Document Number: omg/2003-05-01, May, 2003.
22. A. D. Pace, F. Trilnik, M. Campo. “Assisting the Development of Aspect-Based Multi-Agent Systems Using the Smartweaver Approach.” In: "Software Engineering for Large-Scale Multi-Agent Systems.” Springer, LNCS 2603, March 2003, pp. 165-181.
23. R. Prieto-Diaz, G. Arango. Domain Analysis and Software Systems Modeling. IEEE Computer Society Press, 1991.
24. S. Shavor, J. D’Anjou, S. Fairbrother, et all. The Java Developer’s Guide to Eclipse. Addison-Wesley, 2003.
25. V. Silva, et al. “Taming Agents and Objects in Software Engineering.” In: "Software Engineering for Large-Scale Multi-Agent Systems.” Springer, LNCS 2603, March 2003, , pp. 1-26.
26. P. Tarr, et al. “N Degrees of Separation: Multi-Dimensional Separation of Concerns.” Proceedings of the 21st International Conference on Software Engineering, May 1999.
27. M. Wooldridge, P. Ciancarini (Eds.). “Agent-Oriented Software Engineering: The State of the Art.” In: Agent-Oriented Software Engineering, Springer, LNAI, 2001.
28. M. Wooldridge, N. Jennings, D. Kinny. The Gaia Methodology for Agent-Oriented Analysis and Design. Journal of Autonomous Agents and MASs, Vol. 3, 2000, pp. 285–312.

A Social-Driven Design of e-Business System

Manuel Kolp, T. Tung Do, and Stéphane Faulkner

IAG – ISYS Information Systems Research Unit, University of Louvain,
1, Place des Doyens, Louvain-la-Neuve, Belgium
{kolp,do,faulkner}@isys.ucl.ac.be

Abstract. In the last few years, software applications have increased in complexity and in stakeholder's expectations principally due to new Internet-centric application areas such as e-business, web services, ubiquitous computing, and peer-to-peer networks. Multi-agent systems (MAS) architectures have gained popularity for developing such software. Unfortunately, despite considerable work in software architecture during the last decade, few research efforts have aimed at truly defining frameworks for agent-based architectural design. Considering that a MAS architecture is conceived as a society of software agents, this paper overviews SKwyRL, a social-driven design framework dedicated to build up agent-based systems. The framework proposes a modern approach based on organizational structures and social patterns to define agent architectures notably in the context of e-business system design.

1 Introduction

The meteoric rise of Internet and World-Wide-Web technologies has created overnight new application areas for enterprise software, such as e-business applications. These areas demand software that is robust, can operate within a wide range of environments, and can evolve over time to cope with changing requirements. Moreover, such software has to be highly customizable to meet the needs of a wide range of users and sufficiently secure to protect personal data and other assets on behalf of its stakeholders.

Not surprisingly, researchers are looking for new software paradigms that cope with such requirements. One promising source of ideas for designing such software is the area of multi-agent systems (MAS) architectures. They appear to be more flexible, modular, and robust than traditional including object-oriented ones. They tend to be open and dynamic in the sense they exist in a changing organizational and operational environment where new components can be added, modified or removed at any time. Research in this area has notably emphasized that a MAS is conceived as a society of autonomous, collaborative, and goal-driven software components (agents), much like social organizations.

Such architectures become rapidly complicated due to the ever-increasing complexity of these new business domains and their human or organizational actors. As the expectations of the stakeholders change day after day, as the complexity of the systems, communication technologies and organizations continually increases in today's dynamic environments, developers are expected to produce architectures that must handle more difficult and intricate requirements that were not taken into account

R. Choren et al. (Eds.): SELMAS 2004, LNCS 3390, pp. 70–84, 2005.
© Springer-Verlag Berlin Heidelberg 2005

ten years ago, making thus architectural design a central engineering issue in modern enterprise information system life-cycle [1].

An important technique that helps to manage this complexity when constructing and documenting such architectures is the reuse of development experience and know-how. Thus, styles and patterns have become an attractive approach to reusing architectural design knowledge. Architectural styles are intellectually manageable abstractions of system structure that describe how system components interact and work together [2]. Design patterns describe a problem commonly found in software designs and prescribe a flexible solution for the problem, so as to ease the reuse of that solution. This solution is repeatedly applied from one design to another, producing design structures that look quite similar across different applications [3].

Taking real-world social structures as metaphors, we propose a set of generic architectural structures [4, 5] in the context of the SKwyRL[1] framework [6, 15], whose aim is to construct and validate a software design process for agent-based e-business systems:

- At the architectural design level, organizational styles inspired from organization theory and strategic alliances will be used to design the overall MAS architecture. Styles from organization theory will describe the internal structure and design of the MAS architecture, while styles from strategic alliances will model the cooperation of independent architectural organizational entities that pursue shared goals.
- At the detailed design level, social design patterns drawn from research on cooperative and distributed architectures, will offer a more microscopic view of the social MAS architecture description. They will define the agents and the social dependencies that are necessary for the achievement of agent goals.

The paper is organized as follows. Section 2 overviews architectural organizational styles, details one of them, the structure-in-5, and applies it to design the architecture of an e-business application. Section 3 presents the social design patterns, details one of them, the broker, and applies them to design in details part of the e-business application. Section 4 overviews the agent oriented e-business system implementation. Finally, Section 5 concludes the paper.

2 Organizational Styles

Software architectures describe a software system at a macroscopic level in terms of a manageable number of subsystems, components and modules inter-related through data and control dependencies [7].

System architectural design has been the focus of considerable research during the last fifteen years that has produced well-established architectural styles and frameworks for evaluating their effectiveness with respect to particular software qualities. Examples of styles are pipes-and-filters, event-based, layered, control loops and the like [2]. Examples of software qualities include maintainability, modifiability, portability etc [1]. We are interested in developing a suitable set of architectural styles for

---

[1] SKwyRL: Social arChitectures for Agent Software Systems EngineeRing (http://www.isys.ucl.ac.be/skwyrl)

multi-agent software systems. Since the fundamental concepts of a Multi-Agent System (MAS) are intentional and social, rather than implementation-oriented, we turn to theories which study social structures for motivation and insights. But, what kind of social theory should we turn to? There are theories that study group psychology, communities (virtual or otherwise) and social networks. Such theories study social structure as an emergent property of a social context. Instead, we are interested in social structures that result from a design process. For this, we turn for guidance, in SKwyRL, to organizational theories, namely *Organization Theory* and *Strategic Alliances*. Organizational Theory (e.g., [8, 9]) describe the internal structure and design of an organization, while Strategic Alliances (e.g., [10, 11, 12, 13]) model the strategic cooperation of independent organizational stakeholders who pursue a set of shared goals.

2.1 Organizational Theory

"An organization is a consciously coordinated social entity, with a relatively identifiable boundary, that functions on a relatively continuous basis to achieve a common goal or a set of goals" [9]. Organization theory is the discipline that studies both structure and design in such social entities. Structure deals with the descriptive aspects while design refers to the prescriptive aspects of a social entity. Organization theory describes how practical organizations are actually structured, offers suggestions on how new ones can be constructed, and how old ones can change to improve effectiveness. To this end, schools of organization theory have proposed models patterns to try to find and formalize recurring organizational structures and behaviors.

In the following, we briefly present organizational styles identified in Organization Theory. The structure-in-5 will be studied in detail in Section 2.3.

The Structure-in-5 style. An organization can be considered an aggregate of five sub-structures, as proposed by Minztberg [8]. At the base level sits the *Operational Core* which carries out the basic tasks and procedures directly linked to the production of products and services (acquisition of inputs, transformation of inputs into outputs, distribution of outputs). At the top lies the *Strategic Apex* which makes executive decisions ensuring that the organization fulfils its mission in an effective way and defines the overall strategy of the organization in its environment. The *Middle Line* establishes a hierarchy of authority between the Strategic Apex and the Operational Core. It consists of managers responsible for supervising and coordinating the activities of the Operational Core. The *Technostructure* and the *Support* are separated from the main line of authority and influence the operating core only indirectly. The Technostructure serves the organization by making the work of others more effective, typically by standardizing work processes, outputs, and skills. It is also in charge of applying analytical procedures to adapt the organization to its operational environment. The Support provides specialized services, at various levels of the hierarchy, outside the basic operating workflow (e.g., legal counsel, R&D, payroll, cafeteria).

The pyramid style is the well-know hierarchical authority structure. Actors at lower levels depend on those at higher levels. The crucial mechanism is the direct supervision from the Strategic Apex. Managers and supervisors at intermediate levels only route strategic decisions and authority from the Strategic Apex to the operating

(low) level. They can coordinate behaviors or take decisions by their own, but only at a local level.

The chain of values merges, backward or forward, several actors engaged in achieving or realizing related goals or tasks at different stages of a supply or production process. Participants, who act as intermediaries, add value at each step of the chain. For instance, for the domain of goods distribution, providers are expected to supply quality products, wholesalers are responsible for ensuring their massive exposure, while retailers take care of the direct delivery to the consumers.

The matrix style proposes a multiple command structure: vertical and horizontal channels of information and authority operate simultaneously. The principle of unity of command is set aside, and competing bases of authority are allowed to jointly govern the workflow. The vertical lines are typically those of functional departments that operate as "home bases" for all participants, the horizontal lines represents project groups or geographical arenas where managers combine and coordinate the services of the functional specialists around particular projects or areas.

The auction style involves competitively mechanisms, and actors behave as if they were taking part in an auction. An auctioneer actor runs the show, advertises the auction issued by the auction issuer, receives bids from bidder actors and ensures communication and feedback with the auction issuer who is responsible for issuing the bidding.

2.2 Strategic Alliances

A strategic alliance links specific facets of two or more organizations. At its core, this structure is a trading partnership that enhances the effectiveness of the competitive strategies of the participant organizations by providing for the mutually beneficial trade of technologies, skills, or products based upon them. An alliance can take a variety of forms, ranging from arm's-length contracts to joint ventures, from multinational corporations to university spin-offs, from franchises to equity arrangements. Varied interpretations of the term exist, but a strategic alliance can be defined as possessing simultaneously the following three necessary and sufficient characteristics:

- The two or more organizations that unite to pursue a set of agreed upon goals remain independent subsequent to the formation of the alliance;
- The partner organizations share the benefits of the alliances and control over the performance of assigned tasks;
- The partner organizations contribute on a continuing basis in one or more key strategic areas, e.g., technology, products, and so forth.

In the following, we briefly present organizational styles identified in Strategic Alliances.

The joint venture style involves agreement between two or more intra-industry partners to obtain the benefits of larger scale, partial investment and lower maintenance costs. A specific joint management actor coordinates tasks and manages the sharing of resources between partner actors. Each partner can manage and control itself on a local dimension and interact directly with other partners to exchange resources, such as data and knowledge. However, the strategic operation and coordina-

tion of such an organization, and its actors on a global dimension, are only ensured by the joint management actor in which the original actors possess equity participations.

The arm's-length style implies agreements between independent and competitive, but partner actors. Partners keep their autonomy and independence but act and put their resources and knowledge together to accomplish precise common goals. No authority is lost, or delegated from one collaborator to another.

The hierarchical contracting style identifies coordinating mechanisms that combine arm's-length agreement features with aspects of pyramidal authority. Coordination mechanisms developed for arm's-length (independent) characteristics involve a variety of negotiators, mediators and observers at different levels handling conditional clauses to monitor and manage possible contingencies, negotiate and resolve conflicts and finally deliberate and take decisions. Hierarchical relationships, from the executive apex to the arm's-length contractors restrict autonomy and underlie a cooperative venture between the parties.

The co-optation style involves the incorporation of representatives of external systems into the decision-making or advisory structure and behavior of an initiating organization. By co-opting representatives of external systems, organizations are, in effect, trading confidentiality and authority for resource, knowledge assets and support. The initiating system has to come to terms with the contractors for what is being done on its behalf; and each co-optated actor has to reconcile and adjust its own views with the policy of the system it has to communicate.

2.3 An Organizational Style in Detail

Figure 1 details the structure-in-5 style using the *i** model [14]. *i** is a graph, where each node represents an *actor* (or software agent in this context) and each link between two actors indicates that one actor depends on the other for some goal to be attained. A dependency describes an “agreement” (called *dependum*) between two actors: the *depender* and the *dependee*. The *depender* is the depending actor, and the *dependee,* the actor who is depended upon. The type of the dependency describes the nature of the agreement. *Goal* dependencies represent delegation of responsibility for fulfilling a goal; *softgoal* dependencies are similar to goal dependencies, but their fulfilment cannot be defined precisely; *task* dependencies are used in situations where the dependee is required.

Actors are represented as circles; dependums – goals, softgoals, tasks and resources – are respectively represented as ovals, clouds, hexagons and rectangles; dependencies have the form *depender* → *dependum* → *dependee*.

For instance in Figure 1, the *Technostructure, Middle Line* and *Support* actors depend on the *StrategicApex* for strategic management. Since the goal *Strategic Management* does not have a precise description, it is represented as a softgoal (cloudy shape). The *Middle Line* depends on the *Technostructure* and *Support* respectively through goal dependencies *Control* and *Logistics* represented as oval-shaped icons. The *Operational Core* is related to the *Technostructure* and *Support* actors through the *Standardisation* task dependency and the *Non-operational Service* resource dependency, respectively.

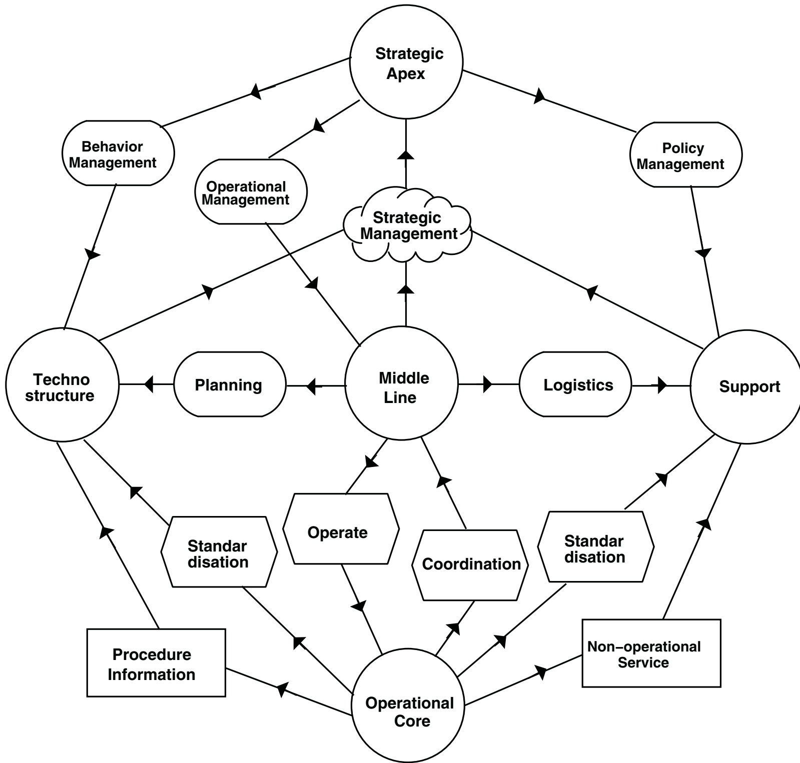


Fig. 1. The Structure-in-5 Style

A number of constraints can also be applied to supplement the basic style:

- The dependencies between the *Strategic Apex* as depender and the *Technostructure*, *Middle Line* and *Support* as dependees must be of type goal;
- A softgoal dependency models the strategic dependence of the *Technostructure*, *Middle Line* and *Support* on the *Strategic Apex*;
- The relationships between the *Middle Line* and *Technostructure* and *Support* must be of goal dependencies;
- The *Operational Core* relies on the *Technostructure* and *Support* through task and resource dependencies;
- Only task dependencies are allowed between the *Middle Line* (as depender or dependee) and the *Operational Core* (as dependee or depender).

2.4 Applying Organizational Styles

This section illustrates how we have applied the structure-in-5 style to design the architecture of a typical E-Business application, called E-Media. It is a typical business-to-consumer application supporting the creation of information sources that facilitate the on-line transaction of products, services, and payments resulting in an effective and efficient interaction among sellers, buyers and intermediaries.

E-Media includes the following main features:

- An on-line web interface allows customers to examine the items in the *E-Media* catalogue, and place orders;
- Customers can search the on-line store by either browsing the catalogue or querying the item database. An online search engine allows customers to search title, author/artist and description fields trough keywords or full-text search;
- Internet communications are supported;
- On-line financial transactions including credit card and anonymity are protected;
- All web information (e.g., product and customer turnover, sales average, …) of strategic importance is recorded for monthly or on-demand statistical analysis;
- Based of this statistical and strategic information, the system permanently manages and adapts the stock, pricing and promotions policy. For example, for each product, the system can decide to increase or decrease stocks or profit margins. It can also adapt the customer on-line interface with new product promotions.

To guide designers in selecting among architectural alternatives, we apply the NFR Framework [17] which offers explicit representation and analysis of non-functional requirements. We identify particular NFRs that can be addressed to characterize MAS architectures; then evaluate the degree to which a given NFR is satisfied by an organizational style; finally select among organizational style alternatives using the propagation algorithms presented in [18].

Figure 2 models the E-Media agent-oriented architecture following the structure-in-5 style.

The Store Front plays the role of the structure-in-5's Operational Core. It interacts with customers and provides them with a usable front-end web application for consulting, searching and shopping media items.

The Back Store constitutes the structure- in-5's Support component. It manages the product database and communicates to the Store Front relevant product information. It stores and backs up all web information about customers, products and sales to be able to produce statistical information (e.g., analyses, average charts and turnover reports). Such kind of information is computed either for a predefined product (when the Coordinator asks it) or on a monthly basis for every product. Based on this monthly statistical information, it provides also the Decision Maker with strategic information (e.g., sales increase or decrease, performance charts, best sales, sales prevision, …).

The Billing Processor plays the role of the structure-in-5's Technostructure in handling customer orders and bills. To this end, it provides the customer with on-line shopping cart capabilities. It also ensures the secure management of financial transactions for the Decision Maker. Finally, it handles, under the responsibility of the Coordinator component, stock orders to avoid shortages or congestions.

As the structure-in-5's Middle Agency, the Coordinator assumes the central position of the architecture. It is responsible to implements strategic decisions for the Decision Maker (Strategic Apex). It supervises and coordinates the activities of the Billing Processor (initiating the stock and pricing policy), the Front Store (adapting the front end interface with new promotions and recommendations) and the Back Store.

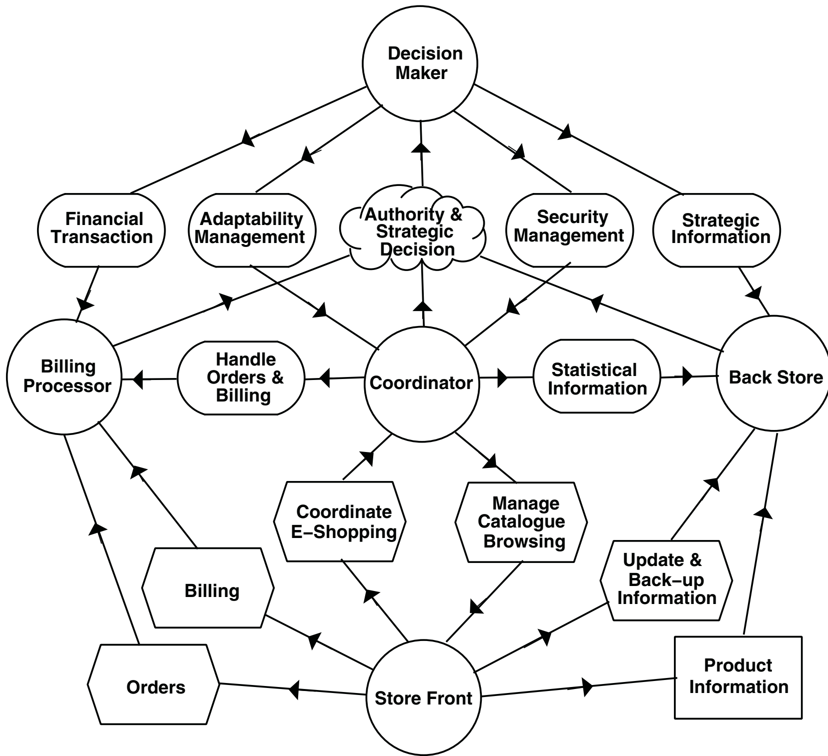


Fig. 2. The E-Media Architecture following the Structure-in-5 Style

3 Social Patterns

The organizational abstraction sketched in the previous sections gives information about the system architecture to be: every time an organizational style is applied, it allows to easily pointing up, to the designer, the required organizational agents.

Next step in MAS architectural design requires detailing and relating identified (organizational) agents to more specific ones in order to proceed with the agent behavior characterization. Namely, each agent in Figure 2 is much closer to the real world system actor behavior than software agent behavior that we consequently aim to achieve. Consequently, once the organizational architectural reflection has figured out the MAS global structure in terms of actors and their intentional relationships, a deepener analysis is required to detail the agent behaviors and their interdependencies necessary to accomplish their roles in the software organization.

To effectively deal with such a purpose, design patterns are used to describe a problem commonly found in software designs and prescribe a flexible solution for the problem, so as to ease the reuse of that solution. In SkwyRL, we adopt social patterns [15] that are design patterns focusing on social and intentional aspects that are recurrent in multi-agent or cooperative systems. Similarly to organizational styles, social patterns are generic structures that define how (a small number of) agents are interacting together in order to fulfill their obligations.

SKwyRL classifies social patterns in two categories. The Pair patterns describe direct interactions between negotiating agents. The Mediation patterns feature intermediate agents that help other agents reach agreement about an exchange of services. These patterns are then applied to design in detail the E-Media application.

3.1 Pair Patterns

The **Booking** pattern involves a client and a number of service providers. The client issues a request to book some resource from a service provider. The provider can accept the request, deny it, or propose to place the client on a waiting list, until the requested resource becomes available when some other client cancels a reservation.

The **Subscription** pattern involves a yellow-page agent and a number of service providers. The providers advertise their services by subscribing to the yellow pages. A provider that no longer wishes to be advertised can request to be unsubscribed.

The **Call-For-**Proposals pattern involves a client and a number of service providers. The client issues a call for proposals for a service to all service providers and then accepts proposals that offer the service for a specified cost. The client selects one service provider to supply the service.

The **Bidding** pattern involves a client and a number of service providers. The client organizes and leads the bidding process, and receives proposals. At every iteration, the client publishes the current bid; it can accept an offer, raise the bid, or cancel the process.

3.2 Mediation Patterns

In the **Monitor** pattern, subscribers register for receiving, from a monitor agent, notifications of changes of state in some subjects of their interest. The monitor accepts subscriptions, requests information from the subjects of interest, and alerts subscribers accordingly.

In the **Broker** pattern, the broker agent is an arbiter and intermediary that requests services from providers to satisfy the request of clients.

In the **Matchmaker** pattern, a matchmaker agent locates a provider for a given service requested by a client, and then lets the client interact directly with the provider, unlike brokers, who handle all interactions between clients and providers.

In the **Mediator** pattern, a mediator agent coordinates the cooperation of performer agents to satisfy the request of a client agent. While a matchmaker simply matches providers with clients, a mediator encapsulates interactions and maintains models of the capabilities of initiators and performers over time.

The **Wrapper** pattern incorporates a legacy system into a multi-agent system. A wrapper agent interfaces system agents with the legacy system (source) by acting as a translator. This ensures that communication protocols are respected and the legacy system remains decoupled from the rest of the agent system.

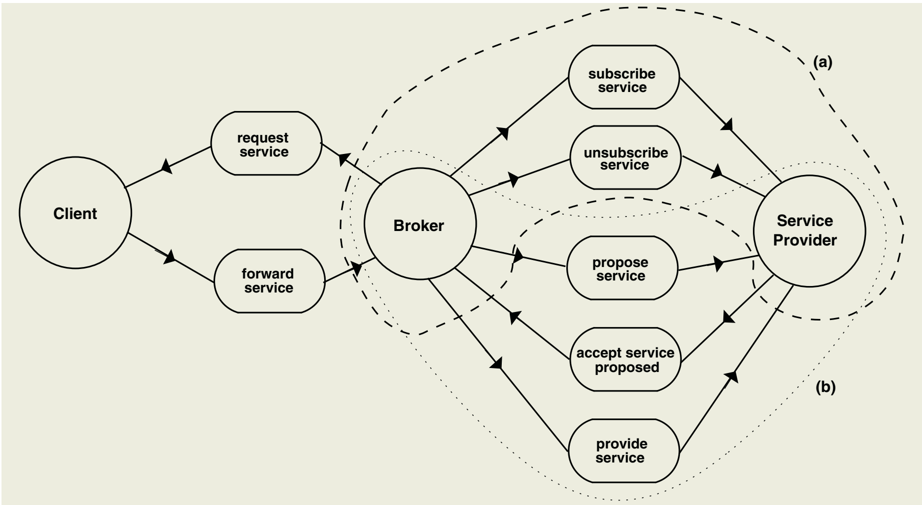


Fig. 3. The Broker Pattern in *i**

3.3 A Social Pattern in Detail

Figure 3 details the Broker social pattern in *i**.

It is considered as a combination of (1) a Subscription pattern (shown enclosed within dashed boundary (a)), that allows service providers to subscribe their services to the Broker agent and where the Broker agent plays the role of yellow-page agent, (2) one of the other pair patterns - Booking, Call-for-Proposals, or Bidding - whereby the Broker agent requests and receives services from service providers (in Figure 3, it is a Call-for-Proposals pattern, shown enclosed within dotted boundary (b)), and (3) interaction between broker and the client: the Broker agent depends on the client for sending a service request and the client depends on the Broker agent to forward the service.

Figure 4 shows a sequence diagram for the Broker pattern. The client (`customer1`) sends a service request (`ServiceRequestSent`) containing the characteristics of the service it wishes to obtain from the broker. The broker may alternatively answer with a denial (`BRRefusalSent`) or a acceptance (`BRAcceptanceSent`).

In the case of an acceptance, the broker sends a call for proposal to the registered service providers (`CallForProposalSent`). The call for proposal (CFP) pattern is then applied to model the interaction between the broker and the service providers. The service provider either fails or achieves the requested service. The broker then informs the client about this result by sending a `InformFailureServiceRequestSent` or a `ServiceForwarded`, respectively.

The communication dimension of the subscription pattern (SB) is given at the top-right and the communication dimension of the call-for- proposals pattern (CFP) is given at the bottom-right part of Figure 4. The communication specific for the broker pattern is given in the left part of the figure.

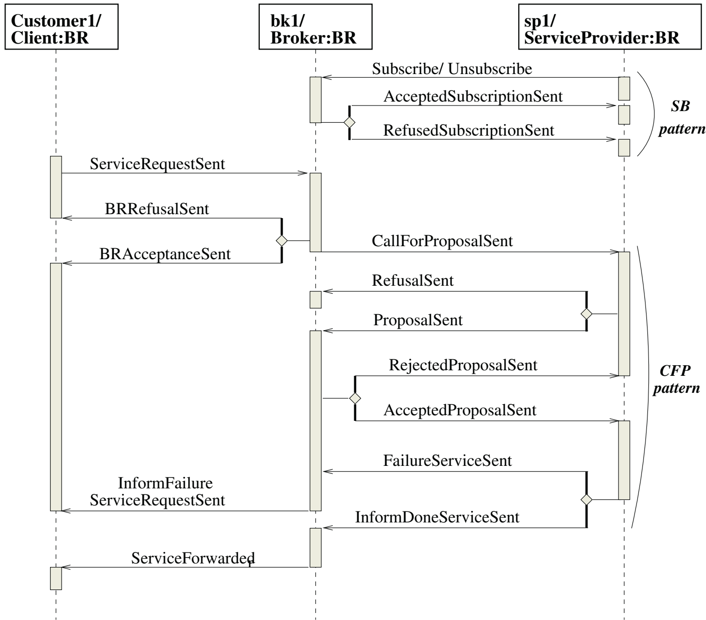


Fig. 4. Interaction Diagram for the Broker Pattern

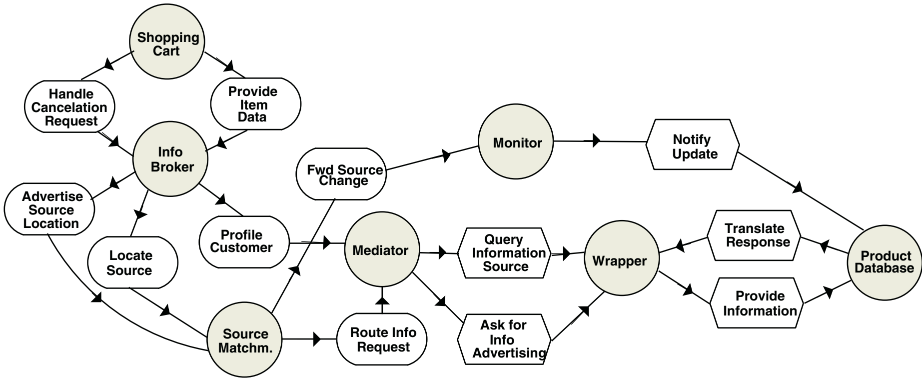


Fig. 5. Decomposing the Store Front with Social Patterns

3.4 Applying Social Patterns

Figure 5 shows a possible use of the patterns for the Store Front component of the e-business system of Figure 2. In particular, it shows how to realize the dependencies *Manage catalogue browsing*, *Update Information* and *Product Information* from the point of view of the Store Front. The Store Front and the dependencies are decomposed into a combination of social patterns [5] involving agents, pattern agents, subgoals and subtasks.

The booking pattern is applied between the *Shopping Cart* and the *Information Broker* to reserve available items. The broker pattern is applied to the *Information*

Fig. 6. E-Media Main Interface

Broker, which satisfies the Shopping Cart's requests of information by accessing the *Product Database*. The *Source Matchmaker* applies the matchmaker pattern to locate the appropriate source for the *Information Broker*, and the monitor pattern is used to check any possible change in the *Product Database*. Finally, the mediator pattern is applied to dispatch the interactions between the *Information Broker*, the *Source Matchmaker*, and the *Wrapper*, while the wrapper pattern makes the interaction between the *Information Broker* and the *Product Database*.

4 Implementation Overview

We briefly describe in this section the e-business system itself by focusing on the role of the *agents* and how they interact. The implementation has been derived from the architectural design explained previously. It has been realized on the JACK agent-oriented development environment [16]

When a user gets connected to E-Media, the Front-Store is instantiated and displays the interface depicted on figure 6. It allows the new coming user to register on the web-site (1). The information provided by the users is handled by the Back-Store which checks the validity (2). Once this has been done, the users can perform purchases on E-Media by adding product to the shopping cart (4). The shopping cart is managed by the Billing-Processor. At any moment during the session the user can use the navigation-bar (3) to switch from one to another section. Promotions (5) and the top 5 best sales (6) are part of the strategic behaviour. The promotion policies are initiated by the Decision-Maker from the strategic information provided by the Back-Store. The Coordinator chooses the best promotions and adapts the promotion interface. The coordinator acts in the same way with the best sales, the Back-Store com-

Fig. 7. E-Media Main Interface, DVD Section

putes the five best sellers and the coordinator is in charge of updating the Front-Store interface.

Figure 7 describes the Store-Front interface when the “DVD” button of the navigation-bar is actived. To start a search the users must fill one or several fields from the search engine (1). The Front-Store sends the query parameters to the Back Store which provides the results to the Front-Store (2). At any moment during the session, if the user clicks on a product (best seller, query result, shopping cart…) a request is sent to Back-Store to provide more information on this product (3).

When the user starts the billing process, the Billing-Processor displays all the items of the shopping *cart* and computes the total and sub-total for each product. Next, it checks the validity of the user Id-Card number. Once the payment is accepted the Billing-Processor informs the Store-Front. A confirmation message is displayed and the shopping cart is cleared.

The E-Media administrator has also the possibility of consulting information computed by the various agents. For instance Figure 8 gives indications on the Billing-Processor. The administrator can either displays the current stock for each product or the orders that have been sent for a certain period.

Particularly for the broker pattern implementation, Figure 9 shows the remote administration tool for the information broker described in Figure 5. The customer sends a service request to the broker asking for buying or selling DVDs. He chooses which DVDs to sell or buy, selects the corresponding DVD titles, the quantity and the deadline (the time-out before which the broker has to realize the requested service). When receiving the customer's request, the broker interacts with the media shops. The interactions between the broker and the media shops are shown in the bottom-right corner of the figure.

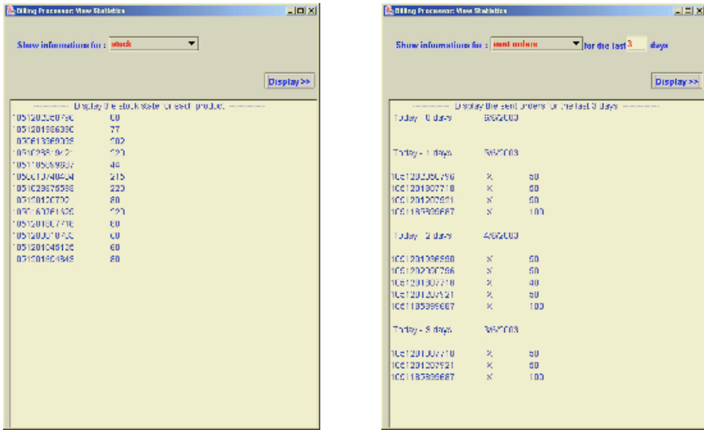


Fig. 8. Statistics on Stock and Sent-Orders

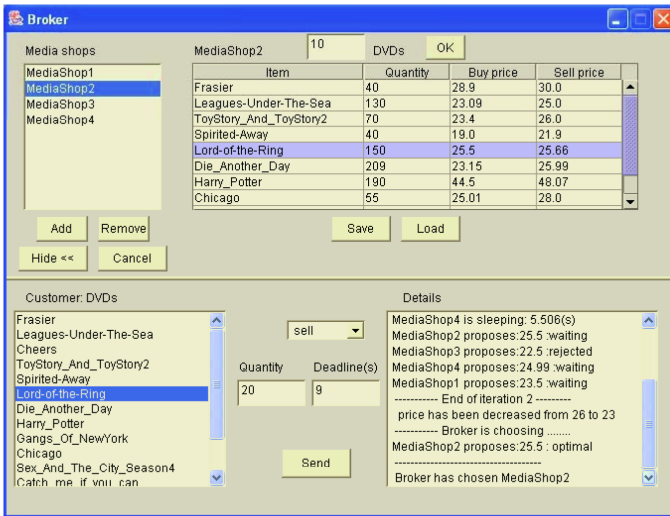


Fig. 9. The Information Broker of E-Media

5 Conclusion

Software engineering for new enterprise application domains such as e-business is forced to build up open systems able to cope with distributed, heterogeneous, and dynamic information issues. Most of these software systems exist in a changing organizational and operational environment where new components can be added, modified or removed at any time. The area of multi-agent systems (MAS) is promising in order to help designing such complex system. Indeed, currently, several agent oriented software methodologies have been proposed.

Unfortunately, architectural design for MAS has not received considerable attention for the past decade. Collection of well-understood architectural styles and patterns exist but for object-oriented rather than agent-oriented systems.

Considering the social intrinsic nature of MAS, this paper has proposed a social-driven framework to design architectures for such systems. The framework considers MAS architectures at two social levels: Organizational architectural styles constitute a macro level; at a micro level it focuses on the notion of social design patterns.

In particular we have detailed and adapted the structure-in-5, a well-understood organizational style used by organization theorists and the Broker social design pattern viewed as a combination of several other social patterns.

The paper has proposed a validation of the framework: it has been applied to develop E-Media, an e-business platform implemented on the JACK agent development environment.

References

1. P. Kruchten: The Rational Unified Process: An introduction. Addison Wesley, 2003.
2. M. Shaw and D. Garlan: Software Architecture: Perspectives on an Emering Discipline. Prentice Hall, 1996.
3. E. Gamma, R. Helm, J. Johnson, and J. Vlissides: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.
4. M. Kolp, P. Giorgini and J. Mylopoulos. “A Goal-Based Organizational Perspective on Multi-Agents Architectures”, Proceedings of the Eighth International Workshop on Agent Theories, architectures, and languages (ATAL’01), 2001.
5. M. Kolp, P. Giorgini, and J. Mylopoulos: “Information systems development through social structures”. Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering (SEKE'02), Italy, 2002.
6. S. Faulkner, M. Kolp, A. Coyette and T. T. Do: “Agent Oriented Design of E-Commerce System Architecture”. Proceedings of the 6th International Conference on Enterprise Information Systems (ICEIS’04), Porto, Portugal, April 2004.
7. L. Bass, P. Clements, and R. Kazman: Software Architecture in Practice. Addison-Wesley, 1998.
8. H. Mintzberg: Structure in fives : designing effective organizations. Prentice-Hall, 1992.
9. W.R. Scott: Organizations: rational, natural, and open systems. Prentice Hall, 1998.
10. M.Y. Yoshino and U. Srinivasa Rangan: Strategic alliances : an entrepreneurial approach to globalization. Harvard Business School Press, 1995.
11. P. Dussauge and B. Garrette: Cooperative Strategy: Competing Successfully Through Strategic Alliances. Wiley and Sons, 1999.
12. J. Morabito, I. Sack, and A. Bhate: Organization modeling: innovative architectures for the 21st century. Prentice Hall, 1999.
13. L. Segil: Intelligent business alliances: how to profit using today's most important strategic tool. Times Business, 1996.
14. E. Yu: Modeling Strategic Relationships for Process Reengineering. PhD thesis, Univesity of Toronto, Department of Computer Science, Canada, 1995.
15. T. T. Do, M. Kolp and A. Pirotte: “Social Patterns for Designing Multi-Agent Systems”. Proceedings of the 15th International Conference on Software Engineering and Knowledge Engineering (SEKE 2003), San Francisco, USA, July 2003.
16. JACK Intelligent Agents. http://www.agent-software.com/.
17. L. Chung: „Representing and Using Non-Functional Requirements: A Process-Oriented Approach”. Ph.D. Thesis, Department of Computer Science, University of Toronto, Canada, 1993.
18. P. Giorgini, J. Mylopoulos, E. Nicchiarelli, and R. Sebastiani: “Reasoning with goal models”. Proceedings of the 21st Int. Conf. on Conceptual Modeling, ER 02, Tampere, Finland, October 2002.

Systematic Integration Between Requirements and Architecture

Lúcia R.D. Bastos[1] and Jaelson F.B. Castro[2]

[1] Banco do Brasil S.A, UF-Tecnologia, Brasília DF, Brazil
luciabastos@bb.com.br
[2] Universidade Federal de Pernambuco, Centro de Informática, Recife PE, Brazil
jbc@cin.ufpe.br

Abstract. Software systems of today are characterized by increasing size, complexity, distribution and heterogeneity. Understanding and supporting the interaction between software requirements and architectures remains one of the challenging problems in software engineering research. The terminology and concepts used for architectural description are quite different from those used for the requirement specification. In spite of this, there is a clear relationship between requirements and architectures. In this chapter we present an approach for integration of system requirements and software architectures within the context of the Tropos project, an information system development framework that is requirement-driven in the sense that it adopts concepts used during early requirements analysis. Our framework advocates that a software system corresponds to the organizational structure, in which actors are members of a group in order to perform specific tasks.

1 Introduction

Requirements Engineering and Software Architecture have become established areas of research, education and practice within the software engineering community. Requirements engineering is concerned with identifying the purpose of the system and the context in which it will be used. Requirements are related to concepts such as goals, conflicts, options and agreements [14]. Moreover, systems characteristics and properties (functional and non-functional) are also described in terms of requirements [7]. Software architectures are important because they represent the particular abstraction for understanding the structure of a system [1], [11]. The software architecture has long been recognized to have a profound impact on the achievement of non-functional goals ("ilities") such as availability, reliability, maintainability, safety, confidentiality, evolvability, and so forth.

Unfortunately, terminology and concepts used for architectural description are quite different from those used for requirement specification. In spite of this, there is a clear relationship between requirements and architectures. Understanding and supporting the interaction between software requirements and architectures remains one of the challenging problems in software engineering research. In this chapter we present an approach for integration of system requirements and software architectures within the context of the Tropos project, an information system development framework that is requirement-driven in the sense that it adopts organizational concepts used during early requirements analysis [5], [6]. Our framework proposes that a soft-

R. Choren et al. (Eds.): SELMAS 2004, LNCS 3390, pp. 85–103, 2005.
© Springer-Verlag Berlin Heidelberg 2005

ware system corresponds to the organizational structure, in which actors are members of a group (software system) in order to perform specific tasks [2], [3]. An organization comprises groups, members, roles and interactions. A member assigned to a role does not work in isolation but interacts and cooperates with other roles.

The extensive use of roles in multi-agent system design emphasizes their importance for complex domains and implementation [9], [17], [21], [32]. Roles can be used both as an intuitive concept in order to analyze requirements in multi-agent systems as well as a behavioural structure in order to implement coherent software architectures.

This chapter is structured as follows. Section 2 presents some basic concepts. Section 3 overviews our approach in context of the Tropos methodology and introduces i* (i-star) requirement models and organizational architectural styles. Section 4 outlines the definitions of our framework**.** Section 5 presents the activities of our process**.** Finally, Section 6 concludes the paper with considerations, related works, contributions and points for further research.

2 Organizational Concepts and Role Theory

The wide variety of subjects and disciplines covered by organizational sciences has given rise to several models and theories on organizations. Role theory can also be of some use as it is widely applied for enterprise modeling, postulating that individuals occupy positions in an organization [4]. In every organization in a society, people play various roles to implement various functions of the system. Roles might be ´president´, ´financial manager´ or ´secretary´ and so on.

Organizations are social groups that are goal-directed and have a set of structured activities to achieve their goals [10]. A social group is a structured set of actor members (sub-groups or agents) that agree on a minimal set of acceptable behaviors. A role is derived through empirical observation of the way that people (actors) work in a particular business setting. This view of an organizational group has a closer alignment to the functioning of the real world. In our approach, organizational concepts are described in terms of the actor members and the roles that they play:

- ***Actor*** is an entity with intentional properties, such as goals, beliefs, abilities and compromises. Actors may be people, software agents or organizational units (group or sub-group). An actor is an entity (e.g., ‘university’ or ‘professor’) that plays one or more roles.
  - ***Group*** is an organization unit (e.g., ‘university’) with a set of organized members being involved in social relationships, pursuing common goals for some period of time with an identifiable domain.
  - **Agent** is a member of a group (e.g., ‘professor’), which plays roles. In this work, agents are system entity, situated in the group environment that is capable of flexible autonomous action in order to meet their goal.
- ***Role*** is an abstract representation of the behavior of actor(s) that perform similar functions in a group, i.e., a *role* denotes a collection of responsibilities (e.g., ‘educator’, ‘adviser’, etc.). Discharging these responsibilities requires the realization of a set of *role responsibilities*. For example, the ‘teacher’ role involves the tasks ‘to teach’ and ‘to supervise’.

- ***Responsibility*** identifies the set of tasks necessary to achieve social objectives (goals) of an actor playing a role.
- ***Goal*** is a condition or state of affairs in the world that the stakeholders would like to achieve.
- ***Task*** specifies a particular way of doing something. Tasks can also be seen as the solutions in the target system, which will satisfy the goals. These solutions provide operations, processes, data representations, structuring and constraints to meet the needs stated in the goals.

An important point to note is the distinction between the actor, i.e. the physical organizational entity, and the *role*, a notion that expresses the responsibility of satisfying certain organizational goal by performing the various tasks within the group. Roles are assigned to actors and summarize a set of skills or capabilities necessary to fulfill a goal. The role is separate from the actor that plays the role. For example, a 'professor' may play multiple roles such as 'educator', 'department chair', etc.

Role in specific contexts of action provide abstract specifications of distributed behavioral patterns. When these roles are performed in a coordinated fashion, they can accomplish a specific objective. Explicit specification of roles, their relations and interdependencies provide a shared context for a group of agents to track the tasks performed.

Others approaches are using roles concepts for modeling social actors in organizational structures [10], social agents in multi-agent system [16], [17], [19], [26], [27], [32], and authority in role-based access control (RBAC) [8]. The extensive use of roles evidences the need for organizational role thinking in multi-agent system requirements [21]. Nevertheless, there is not an agent framework or methodology that provide all abstractions to support organizational groups, roles and interactions in building MAS following an architectural style. In the sequel we consider our approach in context of the Tropos methodology.

3 Tropos Methodology

The Tropos methodology adopts the view of information systems as social structures, that is a collection of social actors, human or software, which acts as agents, positions, or roles and have social dependencies among them [5], [6]. The Tropos methodology spans four phases:

1. Early requirements - concerned with the understanding of a problem by studying an organizational setting; the output is an organizational model that includes relevant actors, their goals and dependencies.
2. Late requirements - the system-to-be is described within its operational environment, along with relevant functions and qualities.
3. Architectural design - the system's global architecture is defined in terms of subsystems, interconnected through data, control and dependencies.
4. Detailed design - behavior of each architectural component is defined in further detail.

The Tropos methodology does not explicitly cover the correlation between requirement elements and architectural elements. Moreover, the information available in the late requirement models is not enough to derive architectural information. In

order to address this issue, the *Systematic Integration between Requirements and Architecture* (SIRA) framework provides a set of elements to specify properties of organizational groups as well as helping to derive architectural properties in the context of the Tropos methodology, as show in Fig. 1.

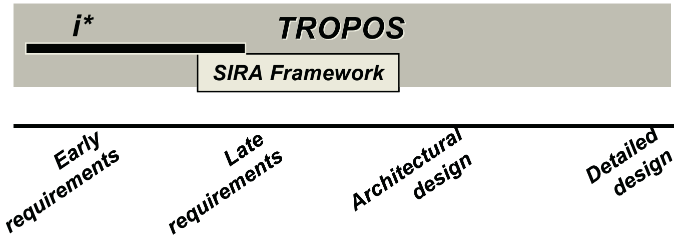


Fig. 1. SIRA Framework in Tropos context

In the following we present a modeling framework for requirements analysis and the organizational-inspired architectural catalogue from Tropos. The requirement models and the architectural catalogue are used as input in the SIRA Process.

3.1 The Requirement Model

The i* (i-star) framework focuses on the modeling of strategic actor relationships of a richer conceptual model of business processes in their organizational settings [28], [29].

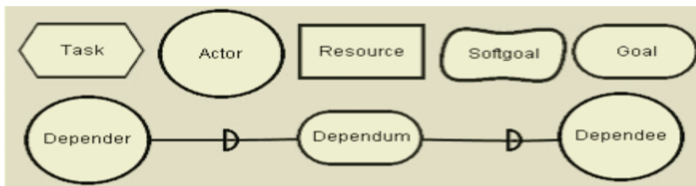


Fig. 2. Elements from the i* framework

The participants of the organizational setting are actors with intentional properties, such as, goals, beliefs, abilities and compromises. A dependency describes an “agreement” (called *dependum*) between two actors playing the roles of *depender* and *dependee,* respectively. The *depender* is the dependent actor, and the *dependee,* the actor who is depended upon. Dependencies have the form *depender→dependum→dependee* (see Fig. 2).

The i* technique consists of two models: Strategic Dependency Model (SD) and Strategic Rationale Model (SR). The Strategic Dependency Model (SD) includes a set of nodes and links connecting them, where nodes represent actors and each link indicates a dependency between two actors. There are four types of dependencies – *goal dependency*, *resource dependency, task dependency* and *soft-goal dependency*. The second model of the i* technique is the Strategic Rationale Model (SR model). It is used to: (i) describe the interests, concerns and motivations of participants in the

process; (ii) enable the assessment of the possible alternatives in the definition of the process; and (iii) research in more detail the existing reasons behind the dependencies between the various actors.

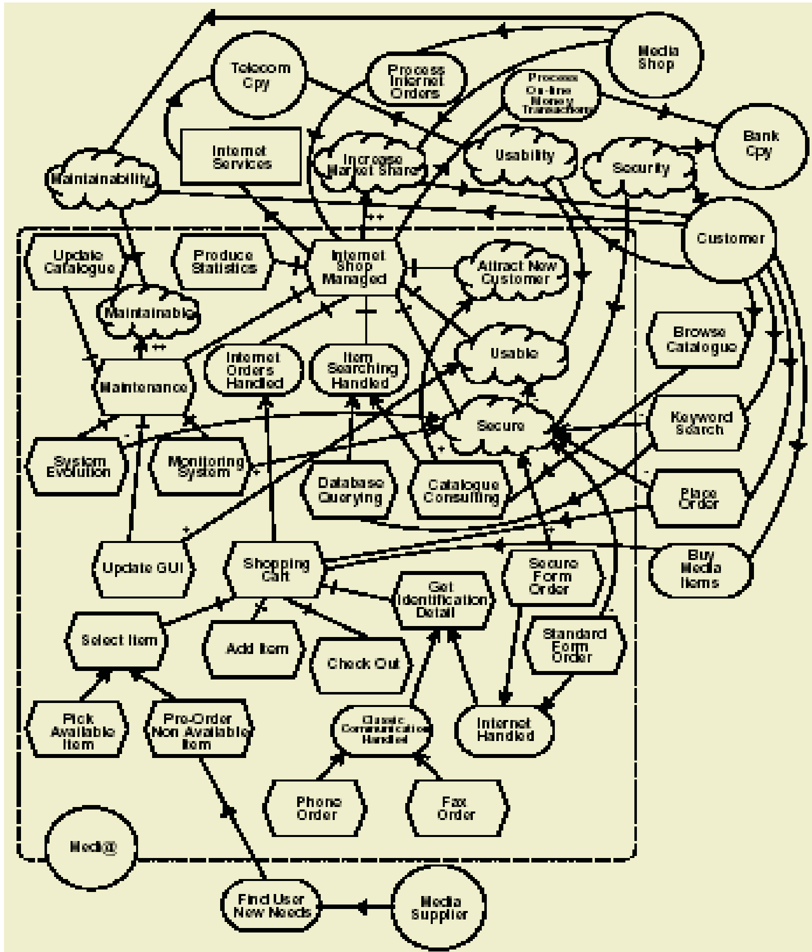


Fig. 3. The SD model of the e-commerce example

In Fig. 3, we have the Strategic Rationale model of an e-commerce example. The *Media Shop* is a store selling and shipping different kinds of media items such as books, newspapers, magazines, audio CDs, videotapes and the like. To increase market share, *Media Shop* has decided to open up a B2C retail sales front on the Internet. The system has been named *Medi@* and is available on the world-wide-web using communication facilities provided by *Telecom Cpy*. It also uses financial services supplied by *Bank Cpy*, which specializes on on-line transactions. The *Medi@* system is introduced as an actor in this strategic dependency model.

The analysis focuses on the software (Media), instead of an external stakeholder. The figure postulates a root task *Internet Shop Managed* providing sufficient support

to the softgoal *Increase Market Share*. That task is firstly refined into goals *Internet Order Handled* and *Item Searching Handled*, softgoals *Attract New Customer*, *Secure* and *Usable* and tasks *Produce Statistics* and *Maintenance*. *Internet Order Handled* is achieved through the task *Shopping Cart*, which is decomposed into subtasks: *Select Item*, *Add Item*, Check *Out,* and *Get Identification Detail*. More details can be founded in [5].

3.2 Architectural Catalogue

System architecture constitutes a relatively small, intellectually manageable model of the system structure, which describes how systems components work together. An architectural style determines the vocabulary of components and connectors that can be used in instances of that style, together with a set of constraints on how they can be combined [11], [24].

*Tropo*s has defined an organizational architectural catalogue for cooperative, dynamic and distributed applications to guide the design of the system architecture [12], [13]. These architectural styles *(pyramid, joint venture, structure in 5, takeover, arm's length, vertical integration, co-optation, bidding)* are based on concepts and design alternatives coming from research on organization management [18].

This work also uses the UML-RT (Unified Modeling Language for Real Time Systems) [23] as notation to represent the organizational architectural catalogue. The notation of capsules, ports and connectors is used to model the architectural actors and their dependencies [25]. Capsules are specialized active classes used for modeling self-contained components of a system.

In Tropos, actors are active entities that carry out actions to achieve goals by exercising their know-how. Hence, an actor is mapped to a capsule. When an actor is a *depender* of some dependency, its corresponding capsule has an implementation port to exchange messages. Ports are physical parts of the implementation of a capsule that mediate the interaction of the capsule with the outside world. A dependency describes an "agreement" (called *dependum*) between two actors playing the roles of *depender* and *dependee,* respectively. A protocol is an explicit specification of the contractual agreement between its participants, which play specific roles in the protocol. Hence, a *dependum* is mapped to a protocol and the *depender* and *dependee* are mapped to protocol roles that comprise the protocol (see Fig. 4).

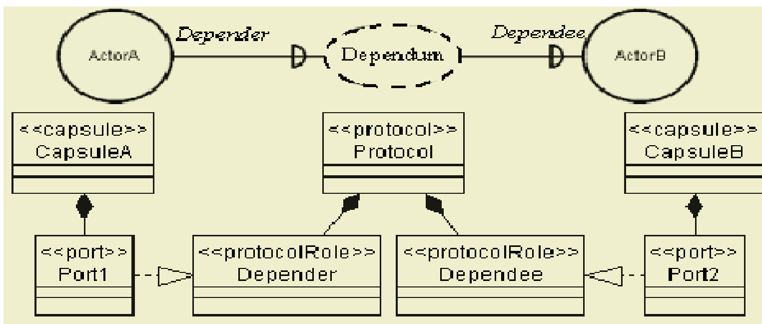


Fig. 4. Mapping dependency between actors to UML-RT

Due to lack of space in this chapter we only detail the *joint venture style* that is a decentralized style based on an agreement between two or more components called principal partners who benefit from operating at a larger scale and reuse the experience and knowledge of their partners. Each principal partner is autonomous on a local dimension and interacts directly with other principal partners to exchange services, data and knowledge. However, the strategic operation and coordination of the joint venture is delegated to a *Joint Management* actor, who coordinates tasks and manages the sharing of knowledge and resources.

Each actor is mapped to a capsule. Each dependency is mapped to a connector. Each dependum, i.e., the agreement between two actors is mapped to protocol. When an actor is a dependee of some dependency, its corresponding capsule has an implementation port (end port) for each dependency (ex. Port1), which is used to provide services for others capsules. When an actor is a depender of some dependency, its corresponding capsule has an implementation port (relay port) to exchange messages (e.g., Port3). This architectural style includes six capsules as shown in Fig.5. The capsule Joint Management is responsible for ensuring the strategic operation and coordination of such a system and its partner capsules on a global dimension. Through the delegation of authority, it coordinates tasks and manages the sharing of knowledge and resources. The two secondary partners are capsules responsible for supplying services or for supporting tasks for the organization core. The three principal partners are capsules responsible for managing and controlling themselves on a local dimension. They can interact directly with other principal partners to exchange, provide and receive services, data and knowledge.

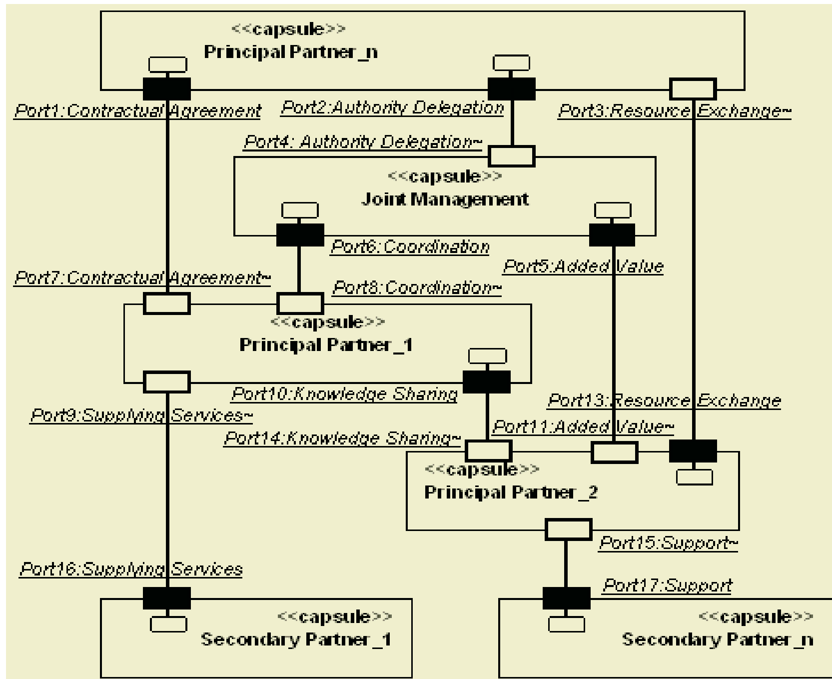


Fig. 5. Using UMLRT collaboration diagram to represent the Joint Venture Style

More details about organizational styles in UML-RT can be found in [25]. The SIRA Framework will be detailed in the next section.

4 SIRA Framework

The SIRA (***S***ystematic ***I***ntegration between ***R***equirements and ***A***rchitecture) framework describes a software system from the perspective of an organization [2], [3]. We advocate that the organizational structure must include the description of organization objectives and the means for their realization. The organizational view extracted from both Strategic Dependency-SD and Strategic Rationale-SR models is used to capture system-related goals.

Fig. 6. The SIRA Framework

The *SIRA* Framework is composed of SIRA Elements and the SIRA Process, as shown in Fig. 6: *SIRA Elements* includes the definitions of organizational Elements (Group and Roles) and Architectural elements. These elements are used to extend the requirement models with organizational information useful for the derivation of architectural models; and the *SIRA Process,* which uses as input the requirements models and architectural catalogue. It generates the *SIRA Elements* and Architectural models, as output. Each SIRA element specification will be detailed below.

4.1 Organizational Elements

This work is concerned with the definition of group and functional roles within the organizational system. The software system will be described as an organizational group structure, in which actors are members of a group in order to perform specific tasks. The organizational structures considers a group as a collection of roles whose behavior co-operatively determines the accomplishment of organizational goals. The organizational group structure is defined in order to represent the social structure of an agent society. The Group (or sub-group) specification is shown in Table 1. Each Group should have a unique name. Moreover, groups are described in terms of their goals, the roles played by their members and norms that apply to the actor tasks.

Table 1. Group

Group	A unique name
Goals	Set of main goals that the group should accomplish.
Roles	Division of labour between the members of a group
Norms	A list of normative expressions that apply to this group

As seen in Table 2, roles have a unique name and are described in terms of objectives to be fulfilled when performing the role. Responsibilities are a list of task ex-

tracted from the set of main goals to be fulfilled by actors, such as “Place order” or “Buy media items” and from interactions between roles. Collaborators are the set of interactions the role has with other roles. Skills are the set of expertise need to perform the role responsibility. Norms are the set of constraint conditions under what the actor should be restricted when performing a role.

Table 2. Role

Role	A unique name
Objective	What an actor of the role is expected to achieve
Responsibilities	A list of task and interactions to be performed
Collaborators	List of roles it interacts
Skills	Set of skills (problem solving knowledge)
Norms	A list of normative expressions that apply to this role (interaction protocol, conflict resolution)

The role interaction protocols are expressed within a particular organizational level, i.e., roles are local to Groups or Sub-Groups. Table 3 shows the specification of an interaction protocol. The protocol is a normative expression on the set of interactions that a role has with other roles.

Table 3. Interaction Protocols

Protocol	A unique name
Objective	Brief description of the nature of the interaction
Initiator	Role(s) responsible for starting the interaction
Responder	Role(s) with which the initiator interacts
Inputs	Information used by the role initiator while enacting the protocol
Outputs	Information supplied by/to the protocol responder during the course of the interactions.

4.2 Architectural Elements

Architectural elements include processing, data and connecting elements. Each architectural element is defined by their properties and relationships [22]. Architecture of a system can be described as a collection of computational components together with a description of the interactions between these components [11]. This chapter uses the terms of *components* and *connectors* to refer to processing and connecting elements, respectively. Architectural elements are:

- ***Components*** – components are the most easily recognized aspect of software architecture. In [22] processing elements are defined as those components that supply the transformation on the data elements. In [11] components are described as the elements that perform computation. A component is an abstract unit of software instructions and internal state that provides a transformation of data via its interface (or ports). A component is defined by its interface and the services it provides to other components.
- ***Connectors*** - A connector is an abstract mechanism that mediates communication, coordination or cooperation among components [24]. Perhaps the best way to think about connectors is to contrast them with components. Connectors enable communication between components by transferring data elements from one interface to another without changing the data.

- ***Constraints*** – The set of architectural constraints of software architecture includes all properties that derive from the selection and arrangement of components, connectors and data within the system. Examples include both the functional properties achieved by the system and non-functional properties, such as relative ease of evolution, reusability of components, efficiency and dynamic extensibility often referred to as quality attributes.

To describe the architectural element information we are using some templates. The template is focus on the logical view of a system including purpose, context and interfaces. Table 4 contains an example of a component specification. Each component should have a unique name. Complex component should be refined into sub-components with the purpose of describe the internal structure of the component. Responsibilities describe the purpose of the component in terms of tasks and interface(s) that it provides. Collaborators describe the other components from which the component requests services in order to achieve its purpose. Some examples of components are illustrated in Fig. 5 (e.g., Joint Management).

Table 4. Component specification

Component/Connector	A unique name	
Responsibilities	Tasks	Interfaces
Collaborators	List of components its interacts	

Interfaces (or ports) are set of interaction points among itself and the external world. They cluster tasks and allow the description of services independent of the components providing them. This is especially convenient if a set of tasks is supported by more than one kind of component so that the set can be referred to in the description of components. It is also very helpful when a component has several responsibilities, each responsibility being expressed by a different interface. Some examples of interfaces are illustrated in Fig. 5 (e.g., Port1: ContractualAgreement).

The SIRA Process will be detailed in the next section.

5 SIRA Process

The SIRA Process consist of three activities (see Fig. 7): The first one, *Analyzing Elements* takes as input the i* Requirement models (together with the Architectural catalogue) to generate the SIRA Elements. The second activity is *Selecting Architecture.* It relies on the use of Non-Functional Requirements framework (NFR) to select architectural styles, based on non-functional requirements extracted from i* models. The third activity, *Relating Elements* links the *SIRA Elements* to architectural elements and generates the architectural model. Each activity will be outlined in the subsections

5.1 Analyzing Elements

The first activity, *Analyzing Elements,* consists of guidelines to identify groups and roles. The organizational context is used to identify the functionality that the system component should provide. Hence, the input to this step is the i* requirements mod-

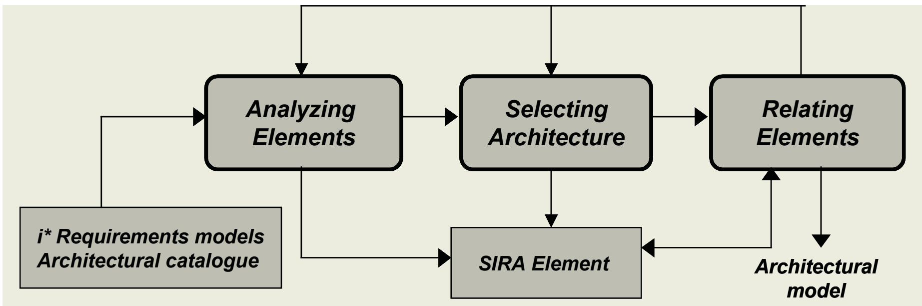


Fig. 7. The *SIRA* Process

els. As output we have the SIRA Elements to complement the information of requirements elements. This activity includes four sub-activities. Each one is used to support the mapping from the i* strategic rationale model into *SIRA-Elements*:

1. ***Identify Groups –*** A group is composed of an actor (or a set of actors) with activities to be performed. The Medi@ system is an example of a Group. It is an e-commerce information software system that supports tasks of the commerce of media items in the Word Wide Web. For example, the main goal of Medi@ actor is to provide service for the other actors such as *Media Shop* and *Customer*. Consequently, the main responsibilities are to fulfil the main goals identified as: "Process Internet Orders", "Buy Media Items" and "Find Users New Needs" (as seen in Fig. 3).
2. ***Identify Goal Refinement –*** The Group responsibilities are identified from the main goal dependencies and task decomposition. Group responsibility involves the refinement of main goals and the distribution of task to be performed.
   Fig. 8 shows the goal refinement with i* means-ends analysis and AND/OR decomposition [05]. In particular means-end analysis aims at identifying tasks, resources and softgoals that provide means for achieving a goal. The goal 'Buy Media Item' is achieved through the tasks 'Consult catalogue', 'Place order' and 'Process order'. The task 'Consult catalogue' is achieved through the tasks 'Browse catalogue' or 'Keyword search'.

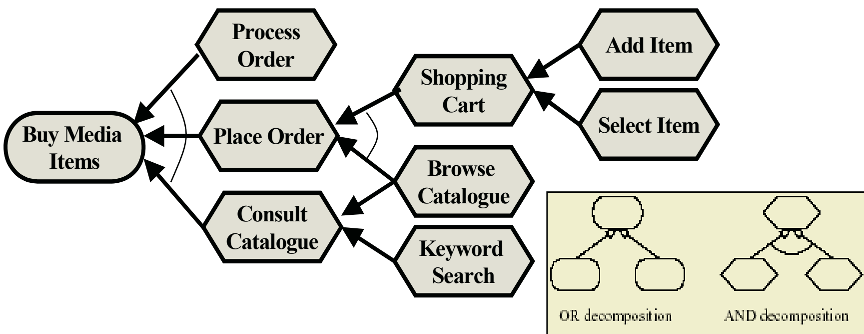


Fig. 8. The partial goal refinement for 'Buy Media Items'

3. ***Identify Roles and Relations –*** Roles are captured in a specific and bounded domain. For example, the e-commerce domain can identify role such as Buyer, Seller or Manager. A role clusters types of behavior into a meaningful unit (role responsibility), which is required to contribute to the group goals. Roles are not isolated. Every role communicates and interacts with other roles. Role responsibility allocation involves the definition and distribution of tasks and interactions to perform a full group goal. If we considering the goal 'Buy Media Item' (see Fig.8), a possible set of roles and responsibilities for Medi@ Group is identified as follows:
   - *Seller Role* is responsible for handling the internet shopping services and for supplying customers (playing the Buyer Role) with a web interface to keep track of items the customer is buying (ex. Consult catalogue and Place order).
   - *Order Provider Role* is responsible for handling the process services that will be executed for a given order (ex. Process Order).
   - *Security Manager Role* is responsible for monitoring and controlling the security check services, suck as: customer profiler and security order form.
   - *Delivery Provider Role* is responsible for interacting with information system of delivery companies.

   Fig. 9 shows the interaction sequence of UML collaboration graph [20] with the roles played to fulfill the group goal 'Buy Media Item' and their interaction protocols. The interaction protocols are expressed within a particular organizational level, i.e., roles are local to Groups or Sub-Groups. Note that UML collaboration diagram interaction semantics provide a precedence relation (i.e., partial order) among interactions. The interaction sequence can be viewed as directed and acyclic graph, with the nodes representing the roles and the arc representing the interaction among them. This representation makes it possible to distinguish between roles that interact and those that do not.

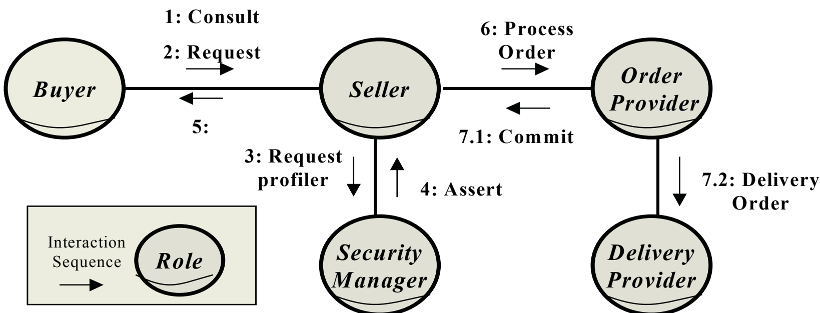


Fig. 9. Role interaction model and a protocol specification between two roles

4. ***Identify Sub-group and Interactions –*** In the most basic form, groups are just a way to refer to a set of roles performing a collaborative set of tasks. Group comprises roles and their interrelations. Thus, identifying Sub-Group concerns deciding on the number and types of roles that should be related, as well as on the number of roles that a group of agents should play. In order to group actor playing roles, this work relies on two organizational structural views: relational and posi-

tional [31]. In the relational view, actor members (or agent) are clustered together based on the strength of the direct relationships with others also referred to as *cohesion*. The positional approach clusters actor members who have similar pattern of relations with others. All those playing a similar role are said to occupy similar structural status positions (see Table 5).

Table 5. Patterns of relationships in organizational structure

	Relational	**Positional**
Clusters based on:	Cohesion	Structural similarity
Belief sharing based on:	Interaction with similar others creates shared beliefs among the clusters members	Playing similar roles creates shared beliefs among those in the same role position

In the e-commerce example, we grouping actors based on structural similarity. The main set of sub-groups is:

1. Store Front sub-group interacts primarily with Customer and provides her with a front-end web application.
2. Back Store sub-group keeps track of all information about customers, products, sales, bills and other strategic important data.
3. Billing Processor sub-group is in charge of the secure management of orders and bills, and other financial data.
4. Manager sub-group manages all of other sub-groups controlling security gaps, availability bottlenecks and adaptability issues.

An important task during architectural design is to select among alternative architectural styles. For each type of alternative, decisions have to be made that, in the end, will produce different architectures.

5.2 Selecting Architecture

Software architectures has been the focus of considerable research, which has resulted in a collection of well-understood architectural styles and a methodology for evaluating their effectiveness with respect to particular software qualities. However, there exist few evaluations of MASs in terms of software qualities. Some software quality attributes for multi-agent architectures were identified from a perspective of organizational styles as architectural styles: predictability, security, adaptability, coordinativity, availability, integrity, modularity, aggregability [12], [13], [30].

A specific refinement, resolution or assignment is selected based on qualitative preferences dictated by positive contributions to high-priority softgoals [7]. This activity involves refining these qualities, represented as softgoals, to sub-goals that are more specific and more precise and then evaluating alternative architectural styles against them, taking into account a specific context (or domain). This SIRA activity includes three sub-activities:

1. ***Identify Architectural Constraints*****:** The input to this sub-activity is a list of requirements, which corresponds to functions, services, constraints or quality attributes. A requirement is an item for which the designer has freedom to choose a solution. A constraint restricts the decision a designer must take. During require-

ments capture, constraints come primarily from the business goals surrounding the system (functional and non-functional requirements). The following quality constraints to select a business-to-consumer architectural alternative could be stated accordingly [6], [12]:

- *Security*: Clients exposed to the internet are, like servers, at risk in web applications. It is possible for web browsers and application servers to download or upload content and programs that could open up the client system to crackers and automated agents.
- *Adaptability:* deals with the way the system can be designed using generic mechanisms to allow web pages to be dynamically changed. It also concerns the catalogue update for inventory consistency.
- *Availability:* Network communication may not be very reliable causing sporadic loss of the server. There are data integrity concerns with the capability of the e-business system to do what needs to be done, as quickly and efficiently as possible in particular with the ability of the system to respond in time to client requests for its services.

2. ***Apply NFR Framework*:** A more precise and systematic analysis of these quality attributes can be done with goal-oriented frameworks such as KAOS [15] or the NFR (non-functional requirements) Framework [7]. In the NFR framework, qualities are represented as *softgoal*s. Analyzing them amounts to a means-ends decomposition of softgoals into more fine-grained subgoals. Each pattern contributes positively/negatively to some of the identified subgoals. As an example, we compare four architectural styles, including some conventional (Pipes & Filter, Layers) [11] and organizational (Structure in 5, Joint Venture) ones. The notations used by the NFR framework includes: +, ++, -, -- to model partial/positive, sufficient/positive, partial/negative and sufficient/negative contributions, respectively.

Table 6. Strengths and weaknesses of 4 architectural styles

	PIPES & FILTERS	***LAYERS***	***S-IN-5***	***JOINT VENTURE***
Security	+	+-	+	+
Availability	+-	+	+	+
Adaptability	-	+-	+-	++

Table 6 summarizes strengths and weaknesses of the four architecture styles with respect to the software quality attributes of Medi@ application. The layered architecture gives precise indications as to the components expected in a business to consumer system. The pipes-and-filters pattern concentrates on the dynamics of input/output data streams. The organizational patterns (Structure-in-5 and Joint Venture) focus on how to organize components expected in an e-business system as well as on the intentional and social dependencies governing these components. More detailed evaluation with respect to the three agent software quality attributes (Security, Adaptability and Availability) can be found in [12].

3. **Select Architectural Style:** The choice of architectural style is based on the application domain. Once the architectural style has been chosen, it should be conform to the application domain functional and non-functional requirements. The result of choosing and refining an architectural style is a collection of (architec-

tural) component types and interactions. However, considering preliminary results from Table 6, we can argue that the organizational architectural styles (*Joint-Venture* or *Structure in 5*) better fit systems and applications that need open and cooperative components such as seen in the e-commerce example. Table 6 suggests the Joint-Venture architectural style as a better solution because it is a more decentralized style.

The next activity will address the mapping among SIRA Elements and architectural components of the selected style.

5.3 Relating Elements

The *Relating Element* activity defines the relationships among requirements and architectural elements, i.e., for example, each joint venture component will be related to each *Sub Group* identified during the *Analyzing Elements* phase.

In order to relate requirements and architecture, an important step is to compare the behavior of the SIRA group and the behavior of a component in a specific architectural style. For a component, typical usage of interfaces is specified while Group specifies how a set of roles interact. Groups and their role interactions should be compatible with components and their interfaces. In the case of incompatibility, the derivation process should return to Analyzing Elements activity and the sub-group may have to be split (or joined). Two different kind of compatibility can be checked:

- Local compatibility – To compare the behavior (tasks and protocols) of the role in a sub-group with the behavior of the corresponding component in the architectural pattern. A Sub-group needs to be compatible with the role it plays in the architectural component.
- Global compatibility – To combine the behavior of all sub-groups and take the intersection with the architectural pattern, i.e. the combination of a set of Sub-Groups should be compatible with the architectural pattern that connects them in a full Group structure.

This SIRA activity is composed of three sub-activities:

1. **Matching Sub-groups and Architectural Styles** – The first step is to check if there exists at least one compatibility between the subgroups, actors/roles (of the group) and components/interfaces (of the architectural style). These sub-groups or actors can become architectural elements (components or agents). In our case study, the selected style was the Joint Venture (Fig. 5). According to this style, the Medi@ software architecture can be decomposed into some autonomous components: a coordination component named Joint Management, and others components identified as Principal Partner_n, Principal Partner_1, Principal Partner_2..., Secondary Partner_1. The Medi@ Group decomposition into Sub-Groups suggest a possible assignment of system responsibilities:
   - The Store Front Sub-Group, with input order responsibilities, can be related to Front Store to provide a customer with a usable front-end web application, which includes a web shopping cart and item browsing.
   - The Provider Sub-group, with order-processing responsibilities, can be related to Billing Processor to support the processing for a given order initialized in Store Front.

- The Manager Sub-group, with managing responsibilities, can be related to Joint Manager to manage controlling security, availability and adaptability.
- The Back Store Sub-Group, with support responsibilities, can be related to Back Store to aggregate functions outside the basic flow of operational tasks, which includes the delivery of items.

2. **Identify Properties of the Architectural Connectors/Interfaces** – Components are bound together by connectors. A connector primarily defines a set of Ports (or interfaces) that ensures connection points through which a component interacts with other. A protocol is an explicit specification of the contractual agreement (obligations) between two components that play specific roles in the protocol. In the Medi@ architecture, all four sub-components need to communicate and collaborate with the system. For instance, Store Front communicates with Billing Processor relevant customer information required to process bills. Back Store organizes, stores and backs up all information coming from Store Front and Billing Processor in order to produce statistical analyses, historical charts and marketing data.
3. **Mapping the Architecture** – A first architectural draft is obtained from the set of sub-groups assigned to functional requirements, identified by roles and interactions. In Fig.10, Medi@ system is decomposed into three principal components (Store Front, Billing Processor and Back Store) controlling themselves on a local dimension and exchanging, providing and receiving services, data and resources with each other. Each of them delegates authority to or is controlled and coordinated by the joint management component (Joint Manager) managing the system in a global dimension.

6 Final Considerations and Related Work

Software systems demand special care in the requirement and architectural modeling phases. A number of goal-based requirement approaches, most notably KAOS [15] and the NFR framework [7], have proposed the explicit use of the notion of 'goals' to structure system requirements and architecture. However, these approaches do not establish an explicit relation between elements of the problem domain and architectural components in solution domain.

Researches in Multi Agent System show that roles can represent system goal and constrain the agent behaviors (ex. Aalaadin [9], MESSAGE [17], MaSE [27] and Gaia [32].

In spite of the significant progress accomplished in the areas of requirement specification and architectural description, we still need frameworks, techniques and tools to support the systematic achievements of the architectural objectives in the complex context of the stakeholders' needs.

In this chapter we present an approach for the integration of requirements and software architectures within the context of the Tropos project. The organizational software architecture is defined at a macro-level. Our approach advocates that a system corresponds to the organizational structure, in which actors are members of a group in order to perform specific tasks. This is an ongoing research and further investigations are still required to evolve this research. In particular, we need to im-

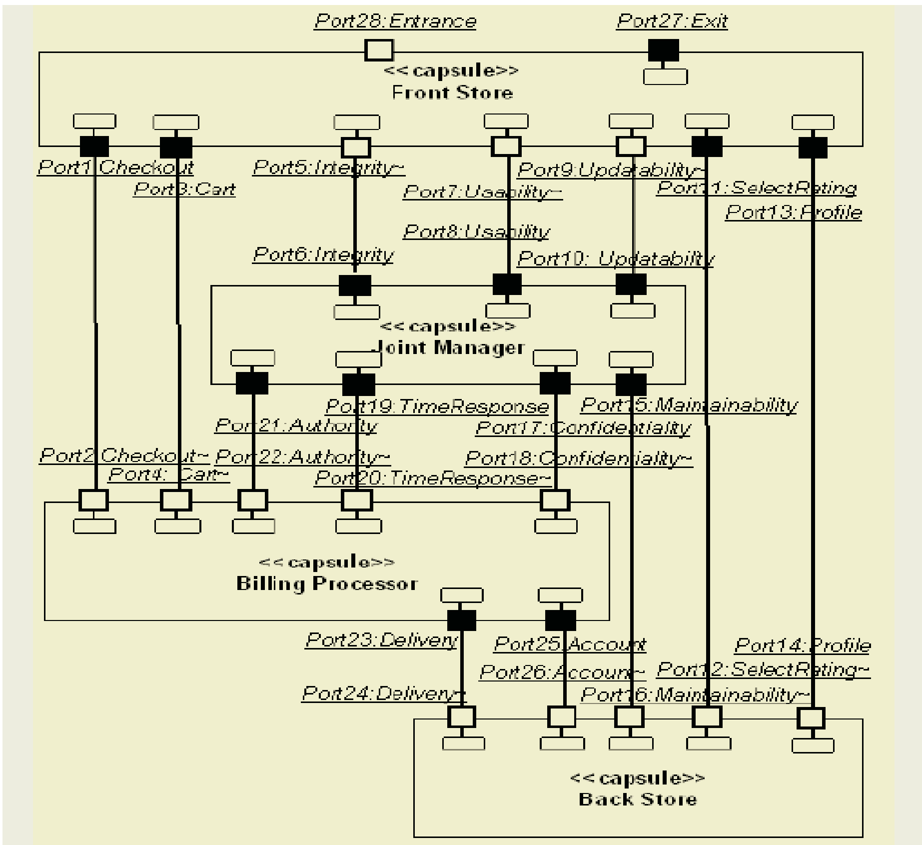


Fig. 10. Medi@ system architecture in UML-RT, the collaboration between capsules occurs within a context defined by protocol implemented by ports that compose each capsule involved in the interaction

prove the architectural derivation rules. Among the main concepts we have: a) to incorporate more detailed rules to analyze the *Group* structures (group, roles and responsibilities); b) to incorporate more systematic rules to match architectural components from *Sub-Groups and Roles*; and c) to apply the proposal to more realistic case studies.

References

1. Bass, L., Clements, P. and Kazman, R. "Software Architecture in Practice". Addison-Wesley, (1998).
2. Bastos, L.R.D. and Castro, J.F.B.: "Integrating Organizational Requirements and Socio-Intentional Architectural Styles". Proceedings of the Second International Workshop From SofTware Requirements to Architectures (STRAW03), 2003. p.114 – 121. In 25th International Conference on Software Engineering 2003 (ICSE'03). Portland, May 2003.

3. Bastos, L.R.D. and Castro, J.F.B.: "Integration between Organizational Requirements and Architecture". Proceedings of the VI Workshop em Engenharia de Requisitos (WER´03). Piracicaba, Brasil, November 2003.
4. Biddle B. J. and Thomas, E. J.: "Role Theory: Concepts and Research". New York: Robert E. Krieger Publishing Company, 1979.
5. Bresciani, P., Giorgini, P., Giunchiglia, F., Mylopoulos, J. and Perini, A.: "TROPOS: An Agent-Oriented Software Development Methodology". In Journal of Autonomous Agents and Multi-Agent Systems, Kluwer Academic Publishers Volume 8, Issue 3, Pages 203 - 236, May 2004.
6. Castro, J., Kolp, M. and Mylopoulos, J.: "Towards Requirements Driven Information Systems Engineering: The Tropos Project". In Information Systems, Vol. 27. Elsevier, Amsterdam, The Netherlands (2002) 365–389.
7. Chung, L., Nixon, B. A., Yu, E. and Mylopoulos, J.: "Non-Functional Requirements in Software Engineering". Kluwer Publishing, 2000.
8. Crook, R., Ince, D. and Nuseibeh, B.: "Towards an Analytical Role Modeling Framework for Security Requirements". Eighth International Workshop on Requirements Engineering: Foundation for Software Quality REFSQ'02. September 2002. Essen, Germany.
9. Ferber, J. and O. Gutknecht.: "A meta-model for the analysis and design of organizations in multiagents systems". In Demazeau, Y., ICMAS' 98, pages 128–135, Paris, 1998.
10. Fox, M., Barbuceanu, M., Gruninger, M. and Lin, J.: "An Organization Ontology for Enterprise Modelling". Simulating Organizations: Computational Models of Institutions and Groups, M. Pritula, K. Carley & L. Gasser (Eds), Menlo Park CA: AAAI/MIT Press, pp. 131-152. 1996.
11. Garlan, D. and Shaw, M.: "An introduction to software architecture". Ambriola & Tortola (eds.), Advances in Software Engineering & Knowledge Engineering, vol. II, World Scientific Pub Co., Singapore, 1993, pp. 1-39.
12. T. T. Do, S. Faulkner and M. Kolp.: "Organizational Multi-Agent Architectures for Information Systems". In Proceedings of the 5th International Conference on Enterprise Information Systems (ICEIS 2003), Angers, France, April 2003.
13. Kolp, M., Giorgini, P. and Mylopoulos, J.: "Organizational Patterns for Early Requirements Analysis". 15th International Conference on Advanced Information Systems Engineering (CAiSE'03), Velden, Austria. June 2003.
14. Lamsweerde, A. van.: "Goal-Oriented Requirements Engineering: A Guided Tour". Proceedings of the 5th International Symposium on Requirements Engineering (RE'01). Toronto, Canadá. August 2001, 249-263.
15. Lamsweerde, A. van.: "Requirements Engineering in the Year 00: A Research Perspective". Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000). Ireland. June 2000.
16. Lupu, E.C. and Sloman, M.: "Towards a role based framework for distributed systems management". Journal of Network and Systems Management, vol. 5, no. 1, P. Press, 1997.
17. MESSAGE: Methodology for Engineering Systems of Software Agents. www.eurescom.de/~pub-deliverables/P900-series/P907/TI1/. Last accessed in June 2004.
18. Mintzberg, H.: "Structure in Fives: Designing effective organizations". P. Hall, 1992.
19. Odell, J., Parunak, H.V.D., and Fleischer, M.: "The Role of Roles in Designing Effective Agent Organizations". Software Engineering for Large-Scale Multi- Agent Systems, Alessandro Garcia et al, LNCS, Springer, 2003.
20. Odell, J., Parunak H. Van D., Bauer, B.: "Representing Agent Interaction Protocols in UML", *Agent-Oriented Software Engineering*, Paolo Ciancarini and Michael Wooldridge eds. Springer-Verlag, Berlin, pp. 121–140, 2001.
21. Partsakoulakis, I., and Vouros, G.: "Roles in MAS: Managing the Complexity of Tasks and Environments". Multi-Agent Systems: An application Science, T. Wagner (eds.), Kluwer Academic, 2004.

22. Perry, D. and Wolf, A.L.: "Foundations for the study of software architecture". ACM SIGSOFT Software Engineering Notes, 17(4), Oct. 1992, pp. 40-52.
23. Selic, B., Gullekson, G. and Ward, P.: "Real Time Object-oriented Modeling". John Wiley & Sons, Inc., 94.
24. Shaw, M. and Clements, P.: "A field guide to boxology: Preliminary classification of architectural styles for software systems". Twenty-First Annual International Computer Software and Applications Conference (COMPSAC`97), Washington, D.C., Aug. 1997, pp. 6-13.
25. Silva, C.T.L.L and Castro, J.F.B.: "Detailing Architectural Design in the Tropos Methodology". 15th Conference on Advanced Information System Engineering (CAISE 03). Klagenfurt, Velden, Austria, 2003.
26. Trzebiatowski, G. L. and Münch, I.: "The Role Concept for Agents in Multi-Agent Systems". Modelling Artificial Societies and Hybrid Organizations. Workshop at KI2001, the Joint German/Austrian Conference on Artificial Intelligence. Vienna, 19-21, 2001.
27. Wood, M.F. and DeLoach, S.A.: "An overview of the Multi-Agent systems engineering methodology". First international workshop on Agent-oriented software engineering (AOSE 2000), p.207-221, January 2001, Limerick, Ireland.
28. Yu, E., and Mylopoulos, J.: "Modeling Organizational Issues for Enterprise Integration". - International Conference on Enterprise Integration and Modeling Technology (ICEIMT'97). Turin, Italy. October 1997.
29. Yu, E.: 'Agent Orientation as a Modeling Paradigm". Wirtschaftsinformatik. 43(2) April 2001. pp. 123-132.
30. Yu, E.: "Modeling Strategic Relationships for Process Reengineering". Ph.D. thesis, Department of Computer Science, University of Toronto, Canada (1995).
31. Zack, M. H.: "Researching Organizational system using social network analysis". Proceedings of the 33rd Hawaii International Conference on System Sciences. *Maui, Hawaii, January 2000* (IEEE 2000).
32. Zambonelli, F., Jennings, N.R., and Wooldridge, M.: "Developing Multi-Agent Systems: The Gaia Methodology". ACM Transactions on Software Engineering and Methodology, 12(3): pp. 317-370. 2003.

Integrating Free-Flow Architectures with Role Models Based on Statecharts

Danny Weyns, Elke Steegmans, and Tom Holvoet

AgentWise, DistriNet
Department of Computer Science K.U.Leuven
Celestijnenlaan 200 A
B-3001 Leuven, Belgium
{Danny.Weyns,Elke.Steegmans,Tom.Holvoet}@cs.kuleuven.ac.be

Abstract. Engineering non-trivial open multi-agent systems is a challenging task. Our research focusses on situated multi-agent systems, i.e. systems in which agents are explicitly placed in a context – an environment – which agents can perceive and in which they can act. Two concerns are essential in developing such open systems. First, the agents must be adaptive in order to exhibit suitable behavior in changing circumstances of the system: new agents may join the system, others may leave, the environment may change, e.g. its topology or its characteristics such as throughput and visibility. A well-known family of agent architectures for adaptive behavior are free-flow architectures. However, building a free-flow architecture based on an analysis of the problem domain is a quasi-impossible job for non-trivial agents. Second, multi-agent systems developers as software engineers require suitable abstractions for describing and structuring agent behavior. The abstraction of a role obviously is essential in this respect. Earlier, we proposed statecharts as a formalism to describe roles. Although this allows application developers to describe roles comfortably, the formalism supports rigid behavior only, and hampers adaptive behavior in changing environments.
In this paper we describe how a synergy can be reached between free-flow architectures and statechart models in order to combine the best of both worlds: adaptivity and suitable abstractions. We illustrate the result through a case study on controlling a collection of automated guided vehicles (AGVs), which is the subject of an industrial project.

1 Introduction

Dealing with the increasing complexity of developing, integrating and managing open distributed applications is a continuous challenge for software engineers. In the last fifteen years, multi-agent systems have been put forward as a key paradigm to tackle the complexity of open distributed applications. In this paper we focus on situated multi-agent systems[1] (situated MASs) as a generic approach

[1] Alternative descriptions are behavior-based agents [4], adaptive autonomous agents [22] or hysteretic agents [16][14].

R. Choren et al. (Eds.): SELMAS 2004, LNCS 3390, pp. 104–120, 2005.
© Springer-Verlag Berlin Heidelberg 2005

to develop self-managing open distributed applications. In particular, we propose an approach that combines aspects of adaptive agent architectures with ideas of rigid modeling of agent behavior for developing these kinds of multi-agent systems.

In situated MASs, agents and the environment constitute complementary parts of a multi-agent world which can mutually affect each other [33]. Situatedness places an agent in a context in which it is able to perceive its environment and in which it can (inter)act. Intelligence in a situated MAS originates from the interactions of the agents in their environment rather than from the capabilities of the individual agents. While interacting, agents form an organization in which they all play and execute their own role(s) in that organization.

The approach of situated MASs has a long history. R. Brooks [4][5] identified the key ideas of *situatedness*, *embodiment* and *emergence of intelligence*. L. Steels [31] and J. L. Deneubourg [11] introduced the basic mechanisms for agents to coordinate through the environment: *gradient fields* and *marks*. P. Maes [22] adopted the early robot-oriented principles of reactivity in a broader context of software MASs. J. Ferber and A. Drogoul [13], M. Dorigo [12], V. Parunak [27] and many other researchers drew inspiration from social insects and adopted the principles in situated MASs. Where the approach of situated MASs started from the rejection of classical agency based on symbolic AI, nowadays the original opposition tends to evolve towards convergence with different schools emphasizing different aspects. The researchers, although having different points of view, are very complementary, and each have their own applications.

Situated MASs have been applied with success in numerous practical applications over a broad range of domains, e.g. manufacturing scheduling [28], network support [3] or peer-to-peer systems [1]. The benefits of situated MASs are well known, the most striking being flexibility, robustness and efficiency.

During the last two years, we developed an agent architecture that enables advanced adaptive agent behavior. The architecture is a hierarchical free-flow architecture which integrates the concept of situated commitments. Situated commitments allow an agent to bias action selection towards actions in its commitments.

Besides the theoretical work on agent architectures, we have been confronted with application engineers who require software engineering support for developing concrete, real-world MASs, the applications include active networking, manufacturing control and supply chain networks. These software engineers require simple and comfortable modeling languages for functionally describing agent behavior. A modeling language based on statecharts resolved this requirement. However, a statechart specification of agent behavior is typically a static, rigid model in that it leaves little room for adaptive and explorative behavior. As a result, the agents in the applications performed behavior that was sometimes unable to adapt to different environmental situations.

Free-flow architectures allow adaptive behavior, yet it is unrealistic to assume that software engineers – starting from the analysis of the problem domain – build a complex free-flow architecture for complex applications, where agents

can perform many actions. For such applications, the architecture quickly becomes unmanageable. We aim to combine the best of both worlds, i.e. the best of adaptive architectures and simple modeling languages. To that end, we retain a flexible action selection mechanism, but complement its description with statecharts. Here, we refrain from considering a statechart description of agent behavior as a kind of sequence chart, but rather use statecharts to describe role composition and to structure related actions within roles only.

This paper is structured as follows. In section 2 we introduce free-flow architectures and give an overview of the statechart formalism we have developed. We discuss problems we encountered when applying them in practice. Section 3, the core of the paper, describes the combined adoption of free-flow architectures and the statechart modeling language. We illustrate our explanation with an example application. Section 4 discusses how the software engineering approach proposed in this paper relates to existing agent-oriented methodologies. Finally, in section 5 we conclude the paper.

2 Free-Flow Architectures and Statechart Models

In this section we start with a brief introduction of free-flow architectures. Then we give a short overview of the statechart formalism we have developed for modeling agent behavior. For both, we point out a number of problems we encountered when applying them in practice. Subsequently we outline an approach to combine free-flow architectures with statechart models.

2.1 Free-Flow Architecture for Adaptive Behavior

Open MASs are characterized by dynamism and change: new agents may join the system, others may leave, the environment may change, e.g. its topology or its characteristics such as throughput and visibility. To cope with such dynamism the agents must be able to adapt their behavior according to the changing circumstances. A well-known family of agent architectures for adaptive behavior are free-flow architectures.

Before we introduce free-flow architectures, we first clarify our perspective on adaptability in this paper. Here we look at adaptability as an agent's ability to deal with different kinds of situations in its environment in a flexible way. We do not look at adaptability in the sense of learning, i.e. as an agent's ability to adjust its behavior in certain kinds of situations over time, according to good or bad experiences of recent choices.

Free-flow architectures are first proposed by Rosenblatt and Payton in [29]. In his Ph.D thesis, T. Tyrrell [32] demonstrated that hierarchical free-flow architectures are superior to flat decision structures, especially in complex and dynamic environments. The results of Tyrrell's work have been very influential, for a recent discussion see [6].

An example of a free-flow architecture is depicted in Fig. 1.

The hierarchy is composed of *activity nodes* (in short nodes) which receive information from internal and external stimuli in the form of *activity*. The nodes

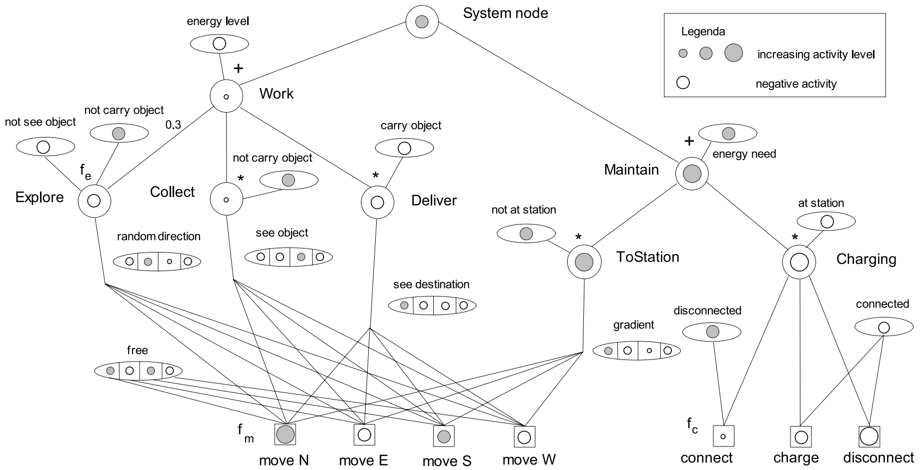


Fig. 1. An example of a free-flow architecture.

feed their activity down through the hierarchy until the activity arrives at the *action nodes* (i.e. the leaf nodes of the tree) where a winner-takes-it-all process decides which action is selected. The main advantages of free-flow architectures are:

– Stimuli can be added to the relevant nodes avoiding the "sensory bottleneck" problem. In a hierarchical decision structure, to make correct initial decisions, the top level has to process most of the sensory information relevant to the lower layers. A free-flow architecture does not "shut down" parts of the decision structure when selecting an action.
– Decisions are made only at the level of the action nodes; as such all information available to the agent is taken into account to select actions.
– Since all information is processed in parallel the agent can take different preferences into consideration simultaneously. E.g. consider an agent that moves to a spotted object but is faced with a neighboring threat. If the agent is only able to take into account one preference at a time it will move straight to the spotted object or move away from the threat. With a free-flow tree the agent avoids the threat *while* it keeps moving towards the desired object, i.e. the agent likely moves around the threat towards a spotted object.

Fig. 1 depicts a free-flow tree of the action selection of a simple robot. This robot lives in a grid world where it has to collect objects and bring them to a destination. The robot is supplied with a battery that provides energy to work. The robot has to maintain its battery, i.e. when the energy level of the battery falls below a critical value the robot has to recharge the battery at a charge station. The left part of the depicted tree represents the functionality for the robot to search, collect and deliver objects. On the right, functionality to maintain the battery is depicted. The *System node* feeds its activity to the *Work* node and the *Maintain* node. The *Work* node combines the received activity

with the activity from the *energy level* stimulus. The "+" symbol indicates that the received activity is summed up. The negative activity of the *energy level* stimulus indicates that little energy remains for the robot. As such the resulting activity in the *Work* node is almost zero. The *Maintain* node on the other hand combines the activity of the *System node* with the positive activity of the *energy need* stimulus, resulting in a strong positive activity. This activity is fed to the *ToStation* and the *Charging* nodes. The *ToStation* node combines the received activity with the activity level of the *not at station* stimulus (the "*" symbol indicates they are multiplied). In a similar way the *Charging* node combines the received activity with the activity level of the *at station* stimulus. This latter is a binary stimulus, i.e. when the robot is at the charge station its value is positive (true), otherwise it is negative (false). The *ToStation* node feeds its positive activity towards the action nodes it is connected with. Each moving direction receives an amount of activity proportional to the value of the *gradient* stimulus for that particular direction. *gradient* is a multi-directional stimulus. The value of this stimulus (for each direction) is based on the sensed value of the gradient field that is transmitted by the charge station. In a similar way, the *Charging* node and the child nodes of the *Work* node (*Explore*, *Collect* and *Deliver*) feed their activity to the action nodes they are connected with. Action nodes that receive activity from different nodes combine that activity according to a specific function. The action nodes for moving actions use a function f_m to calculate the final activity level. A possibility definition of this function is the following:

$$A_{moveD} = \max\ [(A_{Node}\ +\ A_{stimulusD})\ *\ A_{freeD}]$$

Herein is A_{moveD} the activity collected by a move action, D denotes one of the four possible directions, i.e. $D \in \{N, E, S, W\}$. A_{Node} denotes the activity received from a node. The move actions are connected to four nodes: $Node \in \{Explore, Collect, Deliver, ToStation\}$. With each node a particular *stimulus* is associated. $stimulus \in \{random\ direction, see\ object, see\ destination, gradient\}$ are all multi-directional stimuli with a corresponding value for each moving direction. Finally, *free* is a multi-directional binary stimulus that indicates whether the way to a particular direction is free for the robot to move to.

When all action nodes have collected their activity the node with the highest activity level is selected for execution. In the example, the *ToStation* node is clearly dominant over the other nodes connected to actions nodes. Currently the East and West directions are blocked (see the *free* stimulus), leaving the robot two possibilities to move towards the charge station: via North or via South. According to the values of the guiding gradient field, the robot will move northwards, see Fig. 1.

For the simple robot in the example, the free-flow tree is already fairly complex. For a non-trivial agent however, the overall view of the tree quickly becomes very cluttered. When a change is made in one part of such a tree it becomes unclear how this affects the other parts. Although free-flow trees are at best developed with a focus on a particular functionality of the agent, the archi-

tecture itself does not support any structure. From our experiences we learned that it is unrealistic to assume that software engineers build a complex free-flow architecture for complex applications, where agents can perform many actions. For such applications, the architecture quickly becomes unmanageable, it is no longer possible to have an overall view of the architecture.

2.2 Statechart Models

To develop non-trivial open MASs software engineers require suitable abstractions for describing and structuring agent behavior. The abstraction of a role obviously is essential in this respect. Roles are quite general as core abstractions for designing MASs, see e.g. Gaia [9], MESSAGE [8] and also [15][25]. Similar to the definition in [35] we regard a role as an agent's functionality in the context of an organization. Roles provide the building blocks for the social organization of a MAS. Agents are linked to other agents by the roles they play in the organization. The links can be explicit, e.g. a set of agents that pass objects along a chain; or implicit, e.g. in an ant colony a dynamic balance exists between ants that supply the colony with food and ants that maintain the nest.

A number of researchers have proposed state-based approaches to model agent behavior. In SmartAgent [17], UML state machine models are used to model JADE behaviors. [24] points to the strength of statecharts as a constraint mechanism for agent interaction protocols. These and other related work use statecharts to model agent behavior with a focus on inter-agent communication. [18] and [2] are examples in which state machines are used to model reactive behavior. In previous work, we proposed statecharts as a formalism to describe agent behavior, see [19]. In that work we used a statechart formalism to model the behavior of situated agents in terms of roles, with a focus on reusing roles in different applications. Therefore, we extended the standard statecharts notation with new concepts, such as pre-action and post-action. Fig. 2 depicts an example of a role model for a scouting agent.

Although a statechart specification of agent behavior is simple to design and to understand, it is typically a static, rigid model in that it leaves little room for adaptive and explorative behavior. Practical experience with the statechart formalism brought up a number of considerations:

- Action sequences are defined statically. The designer has to enumerate all possible state transitions that can occur, or at least he has to distinguish between discrete categories of environmental situations and corresponding behavioral acts.
- The statechart formalism as developed is in principle only applicable for deterministic agent systems. MASs however, are typically non-deterministic. It is possible to integrate non-determinism in the modeling language, however this would complicate the models significantly. As a result, the agents in the applications performed behavior that was sometime unable to adapt to different environmental situations.

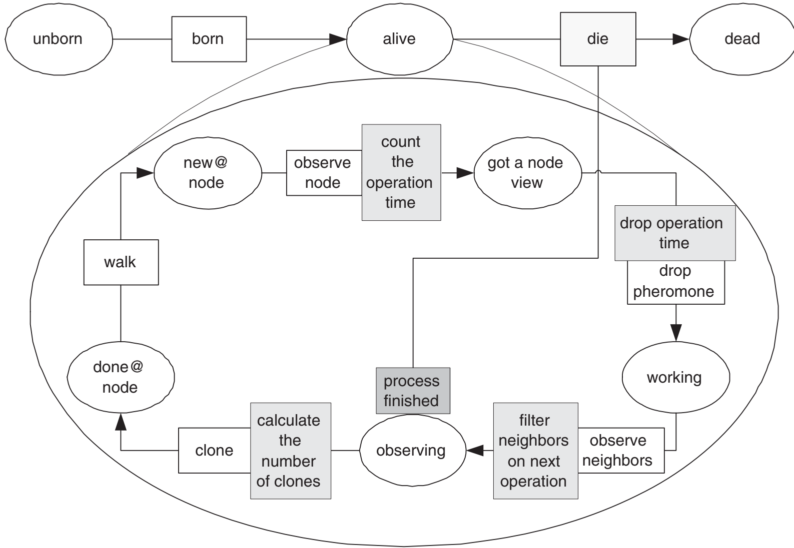


Fig. 2. An example of a statechart model.

– A final remark relates to the set-up of the statechart modeling language. Although different concerns of the agent's behavior can be modelled separately (in terms of building blocks provided by the statechart formalism), different concerns are mixed into one overall diagram. In the proposed statechart formalism no distinction is made between perceptions and actions in the environment, both are modeled as transitions. There is however a fundamental difference between these two activities. For a non-trivial agent merging the two in one model leads to poorly organized models. Another experience relates to the integration of coordination. In [20] we developed inter-agent coordination as a set of pre- and post-actions. The integration of the coordination in the agent's behavior model works well for rather simple agents, however for more complex cases, the models quickly become less surveyable. The underlying problem is that the integration of different concerns should be described separately of the concern descriptions themselves.

Other controlled techniques for engineering agent behavior have been applied such as Petri Nets, see e.g. [23][10][7], however the relationship between these techniques and our statechart modeling approach is out of the scope of this paper.

2.3 Combining the Best of Two Worlds

Agents must be able to adapt their behavior to deal with dynamism and change. Free-flow architectures enable adaptive behavior. However developing free-flow trees for non-trivial agents is a quasi-impossible task for software engineers. Architectures quickly becomes too complex to be manageable. To tackle complexity

suitable abstractions are needed to describe and structure the behavior of the agent. The role statechart modeling language offers a means to this. To combine the best of the two:

1. We extended the free-flow architecture with the abstractions of a role and a situated commitment.
2. We revised the statechart modeling language, i.e. we refrain from considering a statechart description of agent behavior as a kind of sequence chart, but use statecharts to describe role composition and to structure related actions in roles only.

As such statecharts structure the agent behavior reflected in the structure of the free-flow tree. Statecharts also provide an easy way to communicate description of the agent behavior at a higher level of abstraction.

3 Bringing the Statechart Models and Free-Flow Architectures Together

In this section we discuss the combined adoption of statecharts with the extended free-flow architecture. We illustrate our explanation with an example application. We start with a brief introduction of the example application. Next we describe the behavior of the agents with the statechart modeling language. Then we illustrate how the statechart models facilitate the structuring of a free-flow architecture.

3.1 Example Application

In a current research project with an industrial partner we investigate how the paradigm of situated MASs can be applied to the control of logistic machines. Traditional systems use one central controller that instructs the machines to perform jobs based on a calculated plan. The centralized approach lacks flexibility to deal with the increasing demands of adaptability and scalability. By looking at machines as agents of a situated MASs, we aim to convert the centralized control system into a self-managing distributed system, improving adaptability and scalability.

For the case in this paper we limit the discussion to the Automated Guided Vehicle (AGV) transport system. The AGV transport system is typically one, yet a crucial part, of an integral logistic warehouse system. AGV's are unmanned vehicles that transport goods from one place to another. AGV's can supply basic/raw materials to a production department, serve as a link between different production lines or store goods between different processes and connect to the dispatch area.

In a central controlled approach, the functionality of the individual AGV's is rather limited. Each AGV is provided with basic infrastructure to ensure safety, and a typical AGV is able to perform the pick and drop functionality

autonomously. The distribution of jobs, the routing through the warehouse, collision avoidance at junctions etc. are all handled by the central control system.

In this section we look at a number of basic roles for an AGV to deal with jobs autonomously. We take into account functionality for the AGV to find a job, to handle a job, to park when no more work has to be done and finally to ensure that the battery is charged in time.

3.2 AGV's Role Modeling

We distinguish two diagrams and one schema for role modeling. The *role diagram* structures the agent roles and their interdependencies. The *action diagrams* structure the related actions within the roles. Finally the *commitment schema* defines the activation and deactivation conditions for a situated commitment.

Role Diagram. The roles and their interdependencies that describe the behavior of an agent are described in a role diagram. Fig. 3 depicts the role diagram of the AGV's. A role diagram consists of a hierarchy of roles of which some are related through situated commitments.

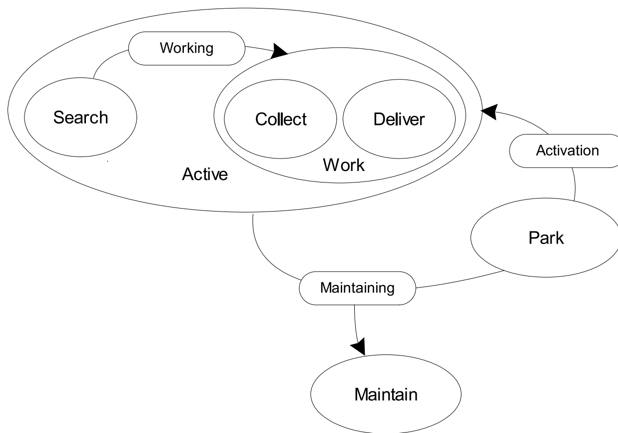


Fig. 3. The role diagram of the AGV.

A *role* is represented by a white oval and the name of the role is written in the oval. A role can consist of a number of sub-roles, and sub-roles of sub-sub-roles etc. As such the role diagram is typically composed as a hierarchy of roles. Roles at the bottom of the hierarchy are called *basic roles*. The first role of the AGV is the *Active* role consisting of two sub-roles, *Search*, i.e. a basic role, and *Work*. The *Work* role is further split up in two sub-roles, *Collect* and *Deliver*, two basic roles. In the *Search* role the AGV searches for a new job. Once the AGV finds a job it will *Collect* the good associated with the job and subsequently *Deliver* the good at the requested destination. Besides the *Active* role, the AGV has the *Maintain* role and the *Park* role. The AGV executes the

Park role when it has no more work to do. In this role the AGV simply moves to the nearest parking place. The *Maintain* role ensures that the AGV keeps its battery loaded. When the energy level crosses a critical value, the AGV finishes its current job and moves towards the nearest charging station. To find its way to the charging station an AGV uses an internal gradient map. At regular time intervals all charging stations broadcast their current status. AGV's use these messages to keep their gradient maps up to date.

A *situated commitment* is represented by a rounded rectangle and the name of the situated commitment is written in the rectangle. A situated commitment is defined as a relationship between one role, i.e. the goal role, and a non-empty set of other roles, i.e. the source roles of the agent. When a situated commitment is activated the behavior of the agent tends to prefer the goal role of the commitment over the source roles. Favoring the goal role results in more consistent behavior of the agent towards the commitment. An agent can commit to itself, e.g. when it has to fulfill a vital task. However, in a collaboration, agents commit relatively to one another, typically through communication. [34] discusses mutual commitments between collaborating agents in detail. In Fig. 3, the *Maintaining* commitment ensures that the AGV maintains its energy level. Since energy is vital for the AGV to function, all roles (except the *Maintain* role of course) are connected as source roles to the *Maintaining* commitment. The *Activation* commitment is activated when the AGV starts to work. This commitment ensures that the AGV remains active once it decides to start working. *Working* is an example of a commitment in a collaboration. The commitment is activated once the AGV accepts a job. This commitment ensures that the AGV acts consistently with the job in progress. As soon as the job is finished, the *Working* commitment is deactivated.

Action Diagram. Action diagrams are defined for the basic roles. An action diagram describes the structure of the related actions for a basic role. In Fig. 4 the action diagram of the *Maintain* role of the AGV is depicted.

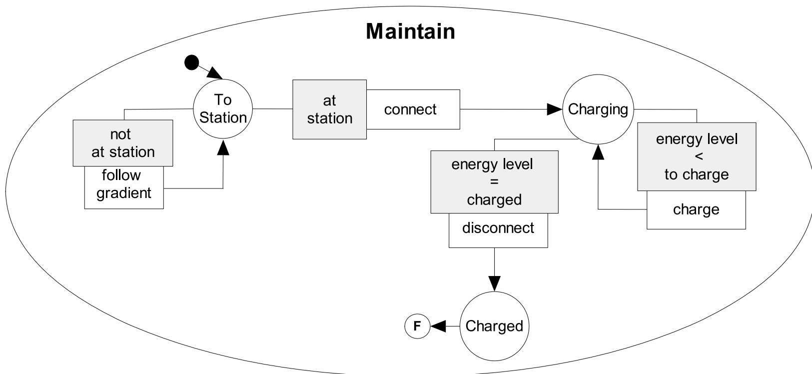


Fig. 4. Action diagram of the Maintain role.

A *state* is represented by a white circle in the diagram. In Fig. 4 three states can be distinguished: *ToStation*, *Charging* and *Charged*. Besides regular states there are two special states. The *default state*, represented by a black circle, indicates the typical start state of the action sequence of the modelled role. On the other hand, there is the *final state*, represented by a circle with an F written in it, that indicates the typical end state of the action sequence of the modelled role. The default and final state are connected to the corresponding regular state via an arrow.

A *transition* connects two states with each other. A transition expresses a change of state due to the execution of an action. An *action*, which is added to a transition, models the functionality that must be performed by an agent to achieve a new desired state from an old state. An action is represented by a white rectangle in which the name of the action is written and which is attached to a transition. To fulfill the *Maintain* role the AGV has to perform four different actions: *follow gradient* to find the charge station, and *connect*, *charge* and *disconnect* to charge the battery (see Fig. 4). The execution of an action may be constrained by a *precondition*. Only when the condition is satisfied the attached action can be executed. A precondition is represented by a gray rectangle in which the precondition is written and which is attached to an action. In Fig. 4 the gray rectangle with *not at station* denotes that the AGV keeps following the gradient until it reaches the charge station. At that time the precondition *at station* becomes true and that enables the AGV to *connect* to the charge station. As long as the condition *energy level* < *charged level* holds, the AGV keeps charging. Finally when condition *energy level* = *charged level* becomes true, the AGV *disconnects* and that finishes the *Maintain* role.

Commitment Schema. For each situated commitment a commitment schema is defined that describes the source roles and the goal role of the commitment as well as its activation and deactivation conditions. Activation and deactivation conditions are boolean expressions based on the internal state of the agent, perceived information or information derived from received messages. Activating situated commitments through communication enable situated agents to setup explicit collaborations in which each participant plays a specific role. In this paper we do not elaborate on this latter scenario, for a detailed discussion we refer to [34]. Fig. 5 depicts the commitment schema for the situated commitment *Maintaining*. This commitment schema expresses that when the *energy level* of the AGV falls below the threshold *to charge* the situated commitment $Maintaining$ is activated. This will urge the AGV to execute the $Maintain$ role over the *Active* and $Park$ roles. Once the battery is recharged the condition *energy level* = *charged* becomes true and that deactivates the $Maintaining$ commitment.

3.3 Free-Flow Architecture

The free-flow tree describes the behavior of the agent in detail. The high-level diagrams for roles and situated commitments described in the previous section

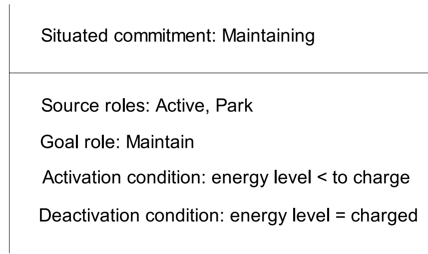


Fig. 5. The commitment schema for the situated commitment Maintaining.

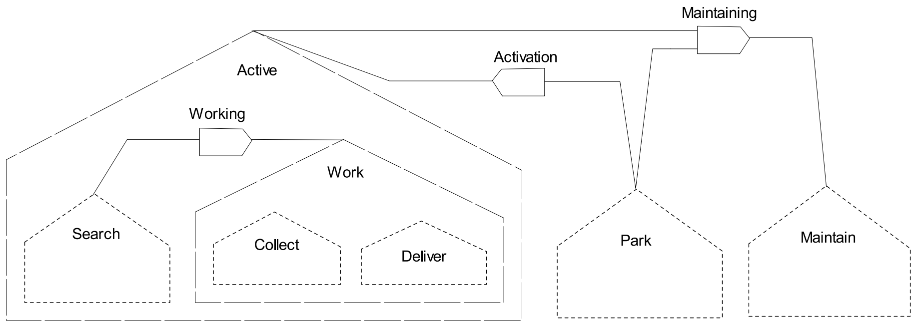


Fig. 6. Skeleton structure of the free-flow tree according to the role diagram of Fig. 3.

serve as a basis for structuring the free-flow tree. The role structure as described in the role diagram (see Fig. 3) is reflected in the skeleton structure of the tree. Fig. 6 depicts the skeleton structure for the AGV example. Roles match to trees in the free-flow tree, sub-roles to sub-trees etc. Situated commitments on the other hand corresponds to connectors that connect the source roles of the situated commitment with the goal role. When a situated commitment is activated extra activity is injected in the goal role relative to the activity levels of the source roles. Details are discussed shortly.

The action diagrams and commitment schemas enable to refine the skeleton tree. Fig. 7 depicts the refined sub-tree for the *Maintain* role and the *Maintaining* commitment.

States in the action diagram correspond to activity nodes in the tree. Preconditions correspond to binary stimuli connected to the corresponding nodes. Examples are the stimuli *at station* or *connected* (compare Fig. 4 and Fig. 7). Each action in the action diagram of the basic role corresponds with an action node in the tree. A number of other non-binary stimuli in the tree represent data in the action diagram that determines the action selection. An example is the stimulus *gradient* that guides the AGV to move towards the station.

The activation and deactivation conditions of the situated commitments, described in the commitment schema, correspond to the conditions associated

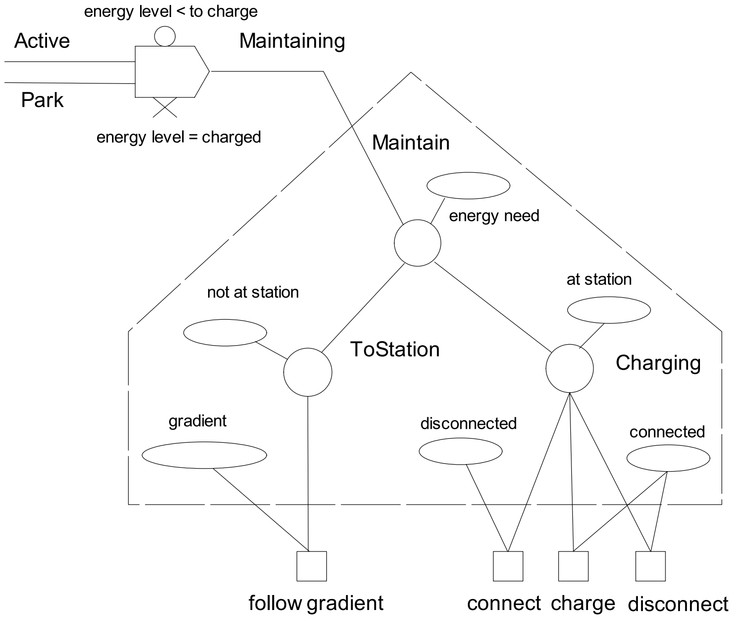


Fig. 7. Refined Maintain role and Maintaining commitment.

with the corresponding connectors in the free-flow activity tree. Fig. 7 illustrates this for the *Maintaining* commitment.

3.4 The Complete Free-Flow Tree

The complete free-flow contains all detailed information needed for action selection. Fig. 8 depicts the completed subtree of the *Maintain* role and the situated commitment *Maintaining*. The abstract action node *follow gradient* in Fig. 7 is refined towards the different moving actions of the AGV. The stimulus *gradient* is split up in a multi-directional stimulus. Each segment represents the tendency (based on the value of the gradient field) of the AGV to move in a particular direction. Besides, a number of extra stimuli represent data that influences the action selection. An example is the multi-directional stimulus *free* that denotes in which direction the AGV is able to drive.

Stimuli needed to verify the activation and deactivation condition are connected to the situated commitment. The *Maintaining* commitment is activated when the value of the *energy level* crosses the threshold *to charge*. The commitment then calculates the extra activity to inject in the *Maintain* role. For the *Maintaining* commitment this extra activity is calculated as the sum ("+" symbol) of the activity level of the *Active* and *Park* role, i.e. the activity levels of the top nodes of these roles. As soon as the battery level reaches the threshold value *charged* the *Maintaining* commitment is deactivated and it then no longer injects extra activity in the *Maintain* role.

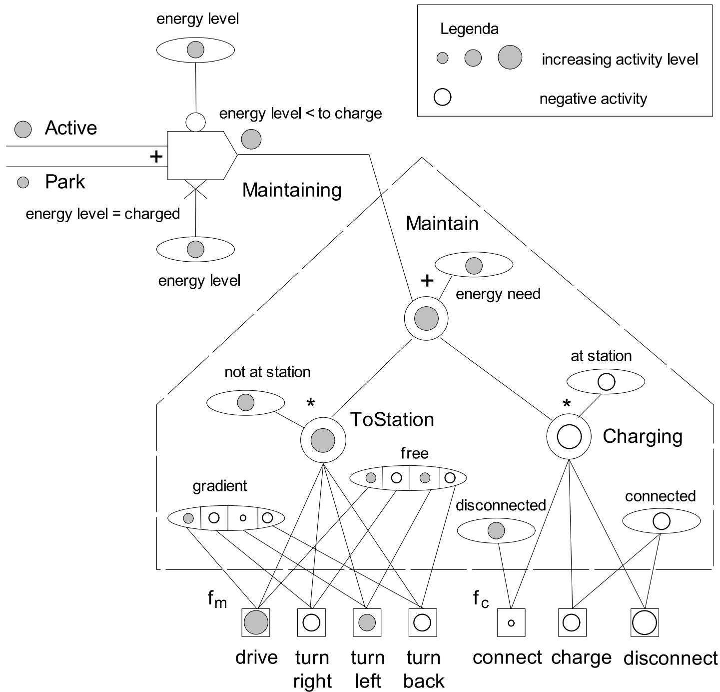


Fig. 8. The complete Maintain role and Maintaining commitment.

4 Discussion

This paper introduces a practical approach to combine adaptiveness of agents and MAS with rigid/controlled engineering. The approach enables engineers to manage the complexity of designing free-flow architectures. The proposed role abstraction allows to represent local agent activity. Roles however, not only "chop up" the behavior of the agent into smaller logical parts, they also introduce a means to enable explicit collaboration between situated agents, reified in situated commitments. In the AGV case e.g., when an *Searching* AGV accepts a job, it activates the *Working* commitment and that will bias the action selection of the AGV towards the *Work* role. As such, the AGV will act consistently towards its commitment in the collaboration, i.e. its agreement to perform the job in progress.

The focus of this paper is mainly on the *integration* of free-flow architectures with role modeling based on statecharts. In ongoing work [30] we described a design process for adaptive agent behavior as part of a multi-agent oriented methodology. This process integrates the engineering approach for behavior design we have proposed in this paper and rigorously describes the subsequent design steps. At the highest level, roles and their interdependencies are caught into a high level model described making use of the statechart modeling language. This model is used as a basis for designing a skeleton of the free-flow

architecture. Next the skeleton is refined such that it contains all details needed for action selection. Finally, the free-flow tree is mapped onto a class diagram that serves as a basis for the implementation of the agent's behavior.

Several agent-oriented methodologies acknowledge the abstraction of a role as a core abstraction for designing multi-agent systems, examples are Gaia [36], MESSAGE [8] or SODA [26]. In these methodologies the design process is described independent of a particular multi-agent architecture, for a recent discussion see Chapter 4 of [21]. When it comes to building a concrete multi-agent application however, the gap between the high level design models and the chosen multi-agent architecture that is used to implement the multi-agent system has to be filled. We aim to bridge this gap enabling designers to build concrete multi-agent systems applications. In particular, the design process described in [30] that builds upon the software engineering approach for behavior design proposed in this paper, enables a designer to incrementally refine the model of the agent behavior from a high level role model toward a concrete agent architecture for adaptive behavior, in casu a free-flow architecture.

5 Conclusion

Engineering software for non-trivial open multi-agent systems is a challenging task. In this paper we proposed a software engineering approach that combines free-flow architectures for adaptive behavior with a statechart modeling language that offers suitable abstractions. Free-flow trees are extended with the abstraction of a role and a situated commitment. The earlier developed statechart formalism is revised and adapted from a rigid description of action sequences towards a description of the role composition of the agent behavior and a structuring of the related actions within the roles. In the paper we illustrated the approach for a case study on controlling a collection of automated guided vehicles.

Currently we are working on a design process for adaptive agent behavior that integrates the engineering approach for behavior design we have proposed in this paper. In future work we intend to extend the design process towards other concerns that need to be engineered in situated MASs such as agent communication and coordination, and the design of the environment of the MAS.

Acknowledgement

The research results presented in this paper have been obtained in the Concerted Research Action on Agents for Coordination and Control - AgCo2 project (K.U.Leuven) and in the Egemin Modular Controls Concept - EMC2 project (Flemish Institute for the Advancement of Scientific-technological Research in the Industry - IWT).

References

1. Babaoglu, O., Meling, H., Montresoret, H.: Anthill: A Framework for the Development of Agent-Based Peer-to-Peer Systems. International Conference on Distributed Computing Systems, Vienna, Austria (2002)
2. Balch, T., Arkin, R.C.: Communication in Reactive Multiagent Robotic Systems. Autonomous Robots **1(1)** (1995)
3. Bonabeau, E., Henaux, F., Guerin, S., Snyers, D., Kuntz, P., Theraulaz, G.: Routing in Telecommunications Networks with Ant-Like Agents. IATA (1998)
4. Brooks, R.: Intelligence without representation. Artificial Intelligence Journal, Vol. 47 (1991)
5. Brooks, R.: Intelligence Without Reason, MIT AI Lab Memo No. 1293 (1991)
6. Bryson, J.: Intelligence by Design, Principles of Modularity and Coordination for Engineering Complex Adaptive Agents. PhD Dissertation, MIT (2001)
7. Cabac, L., Moldt, D.: Formal Semantics for AUML Agent Interaction Protocol Diagrams. 5th International Workshop on Agent-Oriented Software Engineering, AOSE at AAMAS, New York (2004)
8. Caire, G., Leal, F., Chainho, P., et al.: Agent Oriented Analysis Using MESSAGE/UML. Agent-Oriented Software-Engineering II, Lecture Notes in Computer Science, Vol. 2222, Berlin Heidelberg New York, Springer (2001)
9. Cernuzzi, L., Juanand, T., Sterling, L., Zambonelli, F.: The Gaia Methodology: Basic Concepts and Extensions. Methodologies and Software Engineering for Agent Systems, Kluwer (2004)
10. Cost, R.S., Chen, Y., Finin, T., Labrou, Y., Peng, Y.: Modeling agent conversations with colored petri nets. Workshop on Specifying and Implementing Conversation Policies, Seattle, Washington (1999)
11. Deneubourg, J.L., Aron, A., Goss, S., Pasteels, J.M., Duerinck, G.: Random Behavior, Amplification Processes and Number of Participants: How they Contribute to the Foraging Properties of Ants. Physics **22(D)** (1986)
12. Dorigo, M., Gambardella, L.M.: Ant Colony System: A Cooperative Learning Approach to the Traveling Salesman Problem. IEEE Transactions on Evolutionary Computation **1(1)** (1997)
13. Drogoul, A., Ferber, J.: Multi-Agent Simulation as a Tool for Modeling Societies: Application to Social Differentiation in Ant Colonies. Decentralized A.I. **4**, Elsevier North-Holland (1992)
14. Ferber, J.: An Introduction to Distributed Artificial Intelligence. Addison-Wesley (1999)
15. Ferber, J., Gutknecht, O., Michel, F.: From Agents to Organizations: An Organizational View on Multi-Agent Systems. 3th International Workshop on Agent Oriented Software Engineering, AOSE, Melbourne, Australia (2003)
16. Genesereth, M.R., Nilsson, N.: Logical Foundations of Artificial Intelligence, Morgan Kaufmanns (1997)
17. Griss, M.L., Fonseca, S., Cowan, D., Kessler, R.: Using UML State Machine Models for More Precise and Flexible JADE Agent Behaviors. 2th International Workshop on Agent Oriented Software Engineering, AOSE, Bologna, Italy (2002)
18. Harel, D.: Statecharts: A Visual Formalism for Complex Systems. Science of Computer Programming **8(3)** (1987)
19. Holvoet, T., Steegmans, E.: Application-Specific Reuse of Agent Roles. Software Engineering for Large-Scale Multi-Agent Systems, Lecture Notes in Computer Science, Vol. 2603, Berlin Heidelberg New York, Springer (2003)

20. Janssens, N., Steegmans, E., Holvoet, T., Verbaeten, P.: An Agent Design Method Promoting Separation Between Computation and Coordination. Symposium on Applied Computing SAC, Nicosia, Cyprus (2004)
21. Luck, M., Ashri, R., D'Inverno, M.: Agent-Based Software Development. Artech House (2004)
22. Maes, P.: Modeling Adaptive Autonomous Agents. Artificial Life Journal **1(1-2)** (1994)
23. Ferber, J., Magnin, L.: Conception de systemes multi-agents par composants modulaires et reseaux de Petri. Actes des journees du PRC-IA, Montpellier (1994)
24. Odell, J., Parunak, H.V.D., Bauer, B.: Extending UML for Agents. AOIS Workshop at AAAI, www.auml.org (2000)
25. Odell, J., Parunak, H.V.D., Fleisher, M.: The Role of Roles. Journal of Object Technology **2(1)** (2003) http://www.jot.fm/
26. Omicini, A.: SODA: Societies and Infrastructures in the Analysis and Design of Agent-Based Systems. Agent-Oriented Software Engineering, Lecture Notes in Computer Science, Vol. 1957, Berlin Heodelberg New York, Springer (2001)
27. Parunak, H.V.D.: Go to the Ant: Engineering Principles from Natural Agent Systems. Annals of Operations Research **75** (1997)
28. Parunak, H.V.D.: The AARIA Agent Architecture: From Manufacturing Requirements to Agent-Based System Design. Integrated Computer-Aided Engineering **8(1)** (2001)
29. Rosenblatt, K., Payton, D.: A fine grained alternative to the subsumbtion architecture for mobile robot control. International Joint Conference on Neural Networks, IEEE (1989)
30. Steegmans, E., Weyns, D., Holvoet, T., Berbers, Y.: Designing Roles for Situated Agents. 5th International Workshop on Agent-Oriented Software Engineering, AOSE at AAMAS, New York (2004)
31. Steels, L.: Cooperation between distributed agents through self-organization. Proceedings of the First European Workshop on Modeling Autonomous Agents in a Multi-Agent World, Elsevier Science Publishers, Holland (1990)
32. Tyrrell, T.: Computational Mechanisms for Action Selection. Ph.D Thesis, University of Edinburgh (1993)
33. Weyns, D., Holvoet, T.: A Formal Model for Situated Multi-agent Systems. Formal Approaches for Multi-Agent Systems, Special Issue of Fundamenta Informaticae, **63(2–3)** (2004)
34. Weyns, D., Steegmans, E., Holvoet, T.: Protocol Based Communication for Situated Multi-agent Systems. 3th International Joint Conference on Autonomous Agents and Multi-Agent Systems, AAMAS, New York (2004)
35. Wooldridge, M., Jennings, N., Kinny, D.: The Gaia Methodology for Agent-Oriented Analysis and Design. Autonomous Agents and Multi-Agent Systems **3(3)** (2000)
36. Zambonelli, F., Jennings, N.R., Wooldridge, M.: Developing Multiagent Systems: The Gaia Methodology. ACM Transactions on Software Engineering and Methodology **12(3)** (2003)

Aspectizing Multi-agent Systems: From Architecture to Implementation

Alessandro Garcia, Uirá Kulesza, and Carlos Lucena

PUC-Rio, Computer Science Department, LES, SoC+Agents Group,
Rua Marques de São Vicente, 225 - 22453-900, Rio de Janeiro, RJ, Brazil
{afgarcia,uira,lucena}@inf.puc-rio.br
http://www.teccomm.les.inf.puc-rio.br/socagents

Abstract. Agent architectures have to cope with a number of internal properties (concerns), such as autonomy, learning, and mobility. As the agent complexity increases, these agent properties crosscut each other and the agent's basic functionality. In addition, multi-agent systems encompass multiple agent types with heterogeneous architectures. Each of these agent types has different properties, which need to be composed in different ways. In this context, the separation and the flexible composition of agent concerns are crucial for the construction of heterogeneous agent architectures. Moreover the separation of agent concerns needs to be guaranteed throughout the different development phases, especially from the architectural to the implementation phase. Existing approaches do not provide appropriate support for the modularization of agent properties at the architectural stage, and do not promote a smooth transition to the system implementation. This paper presents an aspect-oriented method that allows for a better separation of concerns, supporting the systematic *aspectization* of agent properties through the architectural definition, detailed design and implementation. A multi-agent system for paper reviewing management is assumed as a case study through this paper to show the applicability of our proposal.

1 Introduction

Multi-agent systems (MASs) are composed of heterogeneous agent types with distinct agent properties (concerns), such as adaptation, mobility, collaboration, and learning. The architecture of each agent type in the system incorporates different concerns to be composed in different ways [4, 12]. None is more serious than the difficulty to modularize and compose multiple agent properties [4, 12], requiring a flexible architectural approach. These agent properties are typically overlapping and crosscut the agent's basic functionality [4, 12, 14]. The basic functionalities of agents already are quite complicated, and so agent properties should be designed separately from the agents' basic behaviors [14].

The degrees to which quality requirements (e.g. reusability and maintainability) are met on a MAS are largely dependent on its software architecture [1]. Hence, if an agent architecture that includes suitable support for the separate handling of its multiple properties is chosen from the outset, it is more likely that distinct quality attributes will be achieved throughout the development of multi-agent systems. In addition, the transition of the architectural specification to the detailed design and implementation should be straightforward.

R. Choren et al. (Eds.): SELMAS 2004, LNCS 3390, pp. 121–143, 2005.
© Springer-Verlag Berlin Heidelberg 2005

However, little work has been reported so far on the definition of a development method to structure agent concerns in software systems starting at preliminary development stages. The design of multiple agent concerns with existing architectural approaches [19, 21, 26] usually increases, rather than decreases, their complexity and, consequently, it makes more difficult the task of building high-quality MASs. The perceived low quality of existing multi-agent systems is often attributed to a poor architectural design related to the agent properties [5, 11, 12, 14]. Moreover, software developers usually postpone the modularization of agent properties to the implementation stage. The agent properties are in general introduced into the software system in an ad hoc way [11, 12, 26], leading to agent architectures that are difficult to understand, maintain and reuse.

In this context, this paper presents an aspect-oriented method to support the separation of agent-specific concerns from the architecture to the implementation stages. The contributions are threefold: (i) a set of architectural guidelines to *aspectize* agent concerns on the construction of heterogeneous agent architectures - these architectural steps prescribe solutions independent of programming languages or MAS implementation frameworks [15], (ii) a set of guidelines to enable the detailed design and implementation of aspect-oriented agent architectures, and (iii) a case study of the proposed approach involving a MAS for paper reviewing management.

The basic idea of our proposal is the *aspectization* of agent architectures, using aspect-oriented abstractions to modularize agent-specific concerns at the architectural level. Aspect-oriented software development [9, 10] is an evolving paradigm to modularize concerns, which existing paradigms are not able to capture explicitly. It encourages modular descriptions of complex software by providing support for cleanly separating the basic system functionality from its crosscutting concerns. Aspect is the abstraction used to modularize the crosscutting concerns. However, aspect-oriented approaches have been rarely applied to the MAS domain [6, 12].

The remainder of this paper is organized as follows. Section 2 presents the essential concerns in the development of software agents, and explains why many agent concerns are crosscutting. Section 3 surveys and analyses existing architectural approaches that aim to support the separation of agent-specific concerns. Section 4 presents the notion of aspect-oriented agent architectures and our aspect-oriented method. Section 5 applies the proposed approach to an example. Section 6 discusses the relative advantages and disadvantages of applying the proposed approach. Section 7 discusses related work. Section 8 presents some concluding remarks.

2 Concerns in Agent Architectures

This section presents the essential concerns in the development of software agents. The main concerns are presented in *italic* throughout the section. A concern is some part of a MAS that we want to treat as a single conceptual unit [29]. Agent concerns are modularized throughout software development using different abstractions provided by languages, methods and tools.

A MAS is composed of a set of entities. These entities comprise different types of *agents* and *objects* that are immersed in *environments* [30]. Both objects and agents provide *services* to their clients. However, objects are non-autonomous entities that represent passive system elements. An agent is an interactive, adaptive, autonomous

entity that acts on the environment and manipulate objects [30-33]. As a consequence, the internal architecture of a software agent includes special concerns, which are classified in two categories: *agenthood concerns* (Section 2.1) and *additional concerns* (Section 2.2).

2.1 Agenthood Concerns

Agenthood concerns are the features incorporated by all the agent architectures independently from the agent type. Agenthood usually consists of the *basic agent concerns* – the agent services and the knowledge - and some behavioral properties. Although there is no widely accepted definition of agenthood, autonomy, interaction, and adaptation are considered agenthood properties of software agents, while collaboration, roles, learning, and mobility are neither necessary nor sufficient conditions for agenthood [30-33].

Knowledge. There are different proposed models for knowledge structuring [34, 35], but the knowledge elements are often expressed by *beliefs*, *goals*, *actions*, and *plans* [26, 34, 35]. This work focuses on such a knowledge-structuring model because many projects consider the belief-desire-intention (BDI) model [34] to be the base line for describing the agent knowledge [19, 21, 26]. The agent's beliefs are knowledge elements that describe information about the agent itself, the environment, and its partners. A goal may be realized through different plans. A plan describes a strategy to achieve an internal goal of the agent, and the selection of plans is based on agent beliefs. Actions and plans are used to implement the agent services.

Interaction. The interaction concern is the agent property that implements the communication with the external environment. The interaction behavior basically consists of *receiving messages* and *sending messages* to other agents through *sensors* and *effectors*. Since an message is received, it is *unmarshaled* and stored in an agent *inbox*. When an agent is performing actions and plans, it needs to send messages to the other agents. A message is sent from a simple action or from a plan. The sent messages are *marshaled* and stored in an *outbox*. Agent messages are structured according an agent communication language (ACL) [36]. Since different agents can use different ACLs, messages are translated to an internal message style used by the agent. The interaction protocol can also implement a *sensory behavior*, which consists of observing events in the environment objects.

Adaptation. The adaptation concern is the agent property that modifies the agent according to external and internal events [37, 38]. There are two kinds of adaptation: *knowledge adaptation* and *behavior adaptation*. They follow the same basic protocol, which consists of observing relevant environmental or internal events, gathering the information needed, selecting and invoking the associated adapters [37]. However, knowledge adaptation results in the modification of some piece of the agent knowledge. The behavior adaptation results in either the plan cancellation or the selection of new plans which should be executed next. Sophisticated adapters include reasoning techniques [37, 38] and planners [37, 38].

Autonomy. Autonomy usually means that an agent has control over its own actions and can act independently of others [31, 39, 40]. To be autonomous, the agent must

[30, 31, 39, 40]: (i) *create* its own goals on the basis of internal and external events, (ii) *make decisions* on goal instantiations, (iii) have its own control threads (*execution autonomy*), and (iv) create proactive goals without direct external intervention (proactiveness). The achievement of proactive goals depends on their *degree of autonomy*. The degree of autonomy increases or decreases according to successes and failures of agent actions in the past. These are the dimensions of agent autonomy commonly found on the literature [30, 31, 39, 40].

2.2 Additional Concerns

In addition to the agenthood concerns, the agent developer may have to face additional concerns. The agent may have move from an environment to another, gain new knowledge to improve its performance, and collaborate with other agents.

Mobility. The mobility concern encompasses the behavior to support the agent travels towards remote environments. During the execution of its plans, a mobile agent may have to move from one environment in a network to another in order to achieve its goals. Many facets of the mobility strategy need to be considered [14], including the specification of the *mobile elements*, the descriptions of when the agent should move, the *departure* to remote environments, the *return* to the home environment, and the control of its *itinerary*. In this work, we have focused on weak mobility in which only program code and instance data are moved [16]. In fact, most mobility frameworks support weak mobility [14, 16].

Learning. The learning property involves the agent behavior responsible for refining or gaining knowledge. Cognitive agents learn based on experience as a result of their own actions, their mistakes, the successive interactions with the external environment and the collaborations with other agents [38, 41]. Agents employ different learning techniques, but the general *learning protocol* is the following [38, 41]: (i) an event is detected as relevant, (ii) the event is caught and the information is gathered for the learning purpose, (iii) the learning algorithm processes the gathered information, (iv) the information is stored and alternatively leads to new conclusions, (v) whether a new conclusion is achieved, the knowledge agent is adapted.

Collaboration. Collaboration is viewed as a more sophisticated form of interaction, since it involves collaboration protocols and roles [5]. Collaboration protocols define the ways software agents can interact with other agents in a multi-agent organization. Agents play different *roles* in pursuing their individual goals and work together with other agents in multiple contexts [5]. The role structure is similar to the agent structure. As an agent, a role has *beliefs, goals, actions* and *plans* for carrying out the collaborations with other agents. It may have specific behaviors for interacting with other agents, specific decision algorithms, and specific adaptation strategies. It may also have specialized behaviors related to the additional properties. However, a role cannot exist without an agent.

2.3 Crosscutting Agent Concerns

Several authors have identified that most of the agent properties are often crosscutting, such as mobility [14,54], interaction [4,5], learning [28,53], autonomy [19,44],

and collaboration [14,27]. Some empirical studies confirm their findings [4,7,12]. Fig. 1 shows a partial representation of a multi-agent system [5], which will be used in Section 5 to show the applicability of our proposal. Each set of classes, surrounded by a gray rectangle, has the main purpose of modularizing a specific agent concern, namely interaction, environment, basic concerns, learning, and collaboration.

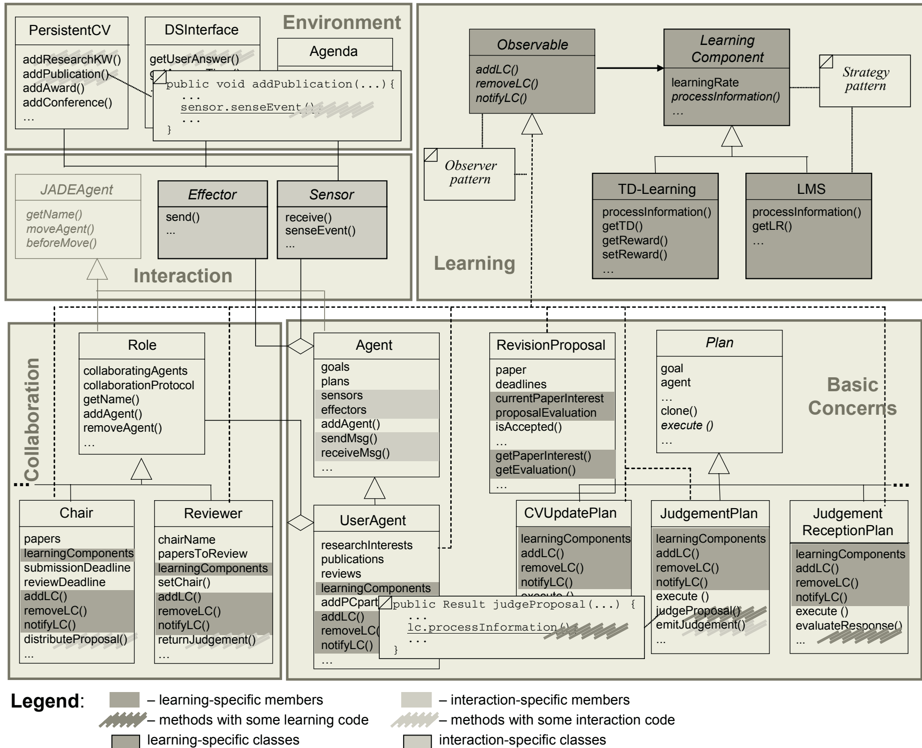


Fig. 1. Crosscutting Agent Concerns

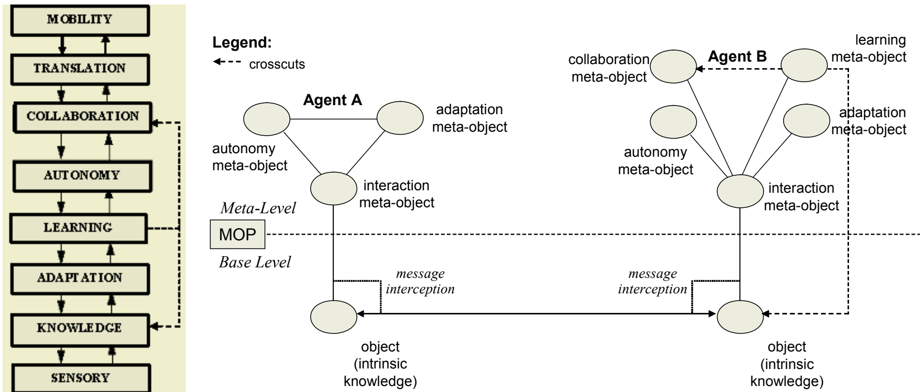
However, note that, for example, the learning concern crosscuts classes implementing other agent concerns; it has a huge impact on the basic agent structure and the collaboration design. Although part of the learning concern is localized in the classes of the Strategy and Observer patterns, learning-specific code replicates and spreads across several class hierarchies of a software agent. Several participants have to implement the observation mechanism and the gathering information and, as a consequence, have learning code in them. Some classes (e.g. RevisionProposal class) have learning-specific knowledge. Adding or removing the learning code from classes requires invasive changes in those classes. Note that even if we try to refactor the object-oriented solution presented in Fig. 1, we cannot find a more modular solution. This problem happens because learning is a crosscutting concern independently of the object-oriented decomposition used [53]. Fig. 1 also illustrates similar problems for the interaction concern, which is usually crosscutting.

3 Existing Architectural Approaches

There are some architectural approaches [19, 21, 26] to promote the separation of agenthood and additional concerns. This section provides a comparison and evaluation of these architectures as well as the identification of the primary limitations of applying them to the separation and integration of crosscutting agent concerns. They rest on traditional architectural patterns, such as the Layers pattern [26], Reflection pattern [19], and the Mediator pattern [21].

Layered Agent Architectures. Kendall et al [26] propose the Layered Agent architectural pattern with multiple layers for the separate representation of the agent concerns (Fig 2a). The interaction concern is modularized in two layers: the translation layer and the sensory layer. The layered architecture establishes a composition style in which all of the interactions feature two-way information flow and only subjacent layers communicate with each other. However, this composition style is very restrictive since agent properties can interact with each other in multiple ways (Section 2.3).

For example, as the agent complexity increases, the learning concern cuts across the different agent layers, such as knowledge and collaboration (Fig 2a). Moreover the evolution of this design approach is cumbersome since removing any of these layers is not a trivial matter; it requires the reconfiguration of the adjacent layers. This layered agent architecture promotes some degree of separation only at the architecture level. When the architectural components – the layers – are decomposed using design patterns, as proposed by the Kendall's approach [26], the architectural separation of agent concerns is degenerated at the detailed design and implementation levels. Previous work has highlighted similar shortcomings of layered architectures [23].



(a) Learning: Crosscutting Layers

(b) Learning: Crosscutting Meta-Objects

Fig. 2. Layered vs. Reflective Agent Architectures

Reflective Agent Architectures. Amandi [19] proposes an architectural approach based on the Reflection architectural pattern [1], called Brainstorm. Reflective software architectures are organized in two levels: the base level that contains the objects, and the meta-level composed of meta-objects. A MOP (meta-object protocol) implements the interface between the base-level and the meta-level. The MOP is responsi-

ble for redirecting the control flow at the base-level to the meta-level in certain execution points of base-level objects. Brainstorm explores meta-objects as abstractions to support the modularization of agent concerns. Each agent concern is modularized in specific meta-objects and associated with based-level objects, which implement the agent's basic concerns (Fig 2b).

Reflective agent architectures improve the separation of agent concerns since meta-objects localize them. However, this architecture introduces some drawbacks. First of all, the composition of multiple meta-objects is not trivial. Meta-objects are objects and their composition rests on inheritance and delegation mechanisms, leading in turn to the problem of crosscutting concerns (Section 2.3). Fig 2b illustrates this problem for the learning concern. Second, a meta-object is often associated with one object. It is very restrictive since several agent properties (meta-objects) can affect directly the basic agent concerns (objects).

Mediator-Based Agent Architectures. The use of mediators is an architectural approach to address the composition of agent concerns that interact in multiple ways. Composition patterns, such as the Mediator pattern [20] and the Composite pattern [20], are mediator-oriented solutions. They provide means of allowing integration of agent properties using a central component, the mediator. The Mediator pattern, for instance, defines a mediator component that encapsulates how a set of components, the colleagues, interact with each other. This solution promotes loose coupling by keeping components from referring to each other explicitly, and it lets the agent developers vary their interaction independently. The Skeleton Agent framework [21] realizes a mediator-based architecture by implementing the Composite pattern. The use of a mediator-based architecture leads to the following problems [4, 6, 7]: (i) the encapsulation of the agent's basic functionality is lost, (ii) agent concerns are scattered and tangled up with each other in the mediator-based design, and (iii) the construction of heterogeneous agent types is difficult as a result of (ii).

4 Aspectizing Software Agents: From Architecture to Implementation

This section overviews aspect-oriented agent architectures [5], which are the foundation of our approach (Section 4.1). This section also presents the proposed guidelines to aspectize MASs. The guidelines are grouped in terms of the different development phases, namely *architecture definition* (Section 4.2), *detailed design* (Section 4.3), and *implementation* (Section 4.4). The guidelines will be applied to an example (Section 5) in order to show the use of our method in practice.

4.1 Aspect-Oriented Agent Architectures

In this paper, the architecture modeling is based on the aSideML language [2], which is a UML extension for representing aspects at different levels of abstraction. Aspects are modular units to encapsulate crosscutting concerns [9, 10]; a*spectual components* (or *architectural aspects*) are aspects [9, 10] at the architectural level. The aSideML language provides two distinct modes for presenting an aspect: (i) condensed or architectural view, and (ii) full view or detailed design view (see Section 4.3). The archi-

tectural view of an aspect suppresses all information about its inner elements. Architectural aspects are UML components represented as diamonds. Each of the aspectual components is related to more than one architectural component, representing their crosscutting nature.

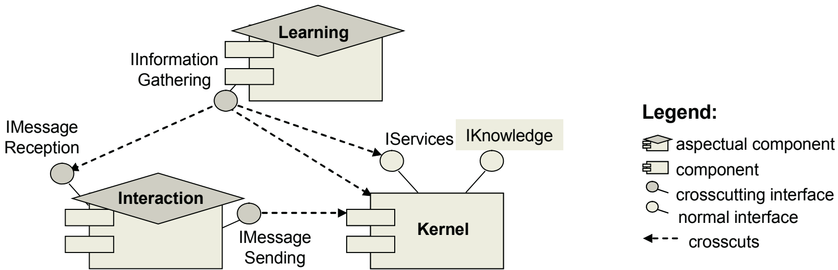


Fig. 3. Aspect-Oriented Architecture

Fig. 3 illustrates some architectural components and their interfaces. Each interface is displayed as a small circle with the interface name placed next to the circle. Each architectural component has one or more interfaces. The interfaces are categorized in two groups: (i) *normal interfaces*, and (ii) *crosscutting interfaces*. A crosscutting interface is different from a normal interface. The latter only provides services to other components. Crosscutting interfaces specify when and how an architectural aspect affects other architectural components. The purpose of crosscutting interfaces is to modularize parts of a concern which usually crosscut other concerns in traditional kinds of decomposition, such as object-orientation (Section 2.3). For example, Fig. 3 shows the IInformation Gathering interface in the Learning component that modularizes the event observation and information gathering, which are issues that usually crosscut the other concerns (Section 2.3). An aspectual component conforms to a set of crosscutting interfaces. Normal interfaces are colored in white and crosscutting ones in gray.

An aspect-oriented agent architecture provides components for *aspectizing* crosscutting agent concerns (Section 2.3). Each agenthood and additional property is modularized as an individual *aspect* [9, 10]. The aspect-oriented architecture is composed of two kinds of architectural components: (i) the *Kernel* component that modularizes the basic agent concerns, and (ii) *aspectual components* (or *architectural aspects*) that separate the crosscutting agent concerns from each other and from the Kernel component. Fig. 3 shows a partial representation of an aspect-oriented agent architecture; it illustrates a Kernel component, two aspectual components, and crosscutting relationships.

The Kernel component implements the services provided for the agent's clients. The Kernel component realizes an interface that makes available services implemented by the agent. This component is also responsible for modularizing the knowledge elements, such as actions, plans, goals, and beliefs. An aspectual component can realize more than one crosscutting interface since it can crosscut multiple agent components in different ways. The interface of an architectural aspect can crosscut the Kernel component and other architectural aspects. An aspect interface crosscuts either internal elements of an agent component or elements of other interfaces. The first case

means that the architectural aspect affects the internal structure or dynamic behavior of the agent component. The second case means that the aspect affects directly an agent architectural aspect.

4.2 Steps for the Architectural Stage

This section presents a set of guidelines to assist software engineers in the design of aspect-oriented agent architectures. The guidelines assist in the configuration of the architectural components and their composition through the specification of normal and crosscutting interfaces in a stepwise fashion. The definition of a crosscutting interface involves the description of the architectural components or interfaces affected by that crosscutting interface. This process determines the relationships between the agent's architectural components, abstracting the internal intricacies of each component.

The steps and substeps should be followed for the architectural definition of each of the system's agents. The following subsections walk through the guidelines to generate the aspect-oriented agent architecture. The steps A1-A4, D1-D4 are mandatory for all agent types since they represent guidelines for dealing with the agenthood concerns. The remaining steps are optional because they comprise the additional concerns.

Step A1. Define the Kernel component.

a) Define the agent's basic interfaces. Each agent can have one or more normal interfaces which make the agent services available to the environment.
b) Define the normal interface for the agent knowledge maintenance.

Step A2. Define the Interaction aspectual component.

a) Define the crosscutting interfaces for the sensory behavior.
b) Define the crosscutting interfaces for message reception.
c) Define the crosscutting interfaces for message sending.

Step A3. Define the Adaptation aspectual component.

a) Define the crosscutting interfaces for knowledge adaptation.
b) Define the crosscutting interfaces for behavior adaptation.

Step A4. Define the Autonomy aspectual component.

a) Define the crosscutting interface for addressing the thread management. This interface specifies the policies for starting and finalizing agent threads. This interface may be not necessary in the cases where the thread is attached to the agent by the enclosing system and not by the application.
b) Define the crosscutting interface for goal creation.
c) Define the crosscutting interfaces for controlling the autonomy degree.
d) Define the crosscutting interfaces for decision making. Reactive agents [38] only need to decide according to external events. Proactive agents [38] make decisions on the basis of both internal and external stimulus.

Step A5. Define the Mobility aspectual component.

a) Define the crosscutting interfaces for specifying the elements to be moved together with the agent.
b) Define the crosscutting interfaces for the agent departure and the agent return.

Step A6. Define the Learning aspectual component.

a) Define the crosscutting interfaces for observing the agent's internal events and gathering the contextual information.
b) Define the crosscutting interfaces for describing the learning-specific knowledge.

Step A7. Define the Collaboration aspectual component.

a) Define the crosscutting interfaces for enforcing the collaboration protocols.
b) For each role, define a Role aspectual component, and:
   a. define the role architecture by starting from Step A1.
   b. define the crosscutting interface for role binding.
   c. define the crosscutting interface for describing the role-specific knowledge.
   d. associate the Role component with the respective protocol interfaces (A7-a).

4.3 Steps for the Detailed Design Stage

Each step in this phase is associated with an architectural step (Section 4.2), refining an architectural component previously defined. A design step has two basic procedures: (i) the refinement of the corresponding architectural component, which is usually decomposed in terms of an abstract aspect, concrete aspects, and/or classes; and (ii) the refinement of the corresponding crosscutting or normal interfaces. The detailed design of a normal interface involves the definition of the services to be made available by the interface.

The detailed design of a crosscutting interface encompasses the specification of the join points, pointcuts, advices, and inter-type declarations. *Join points* are well-defined points in the dynamic execution of the system components. Examples of join points are method calls and method executions. *Pointcuts* have name and are collections of join points. *Advice* is a special method-like construct attached to pointcuts. *Inter-type declarations* introduce attributes, methods, and interface implementation declarations into the components to which the crosscutting interface is attached.

Step D1. Refine the Kernel component.

a) Create a class to represent the agent type. This class should extend a generic, abstract Agent class that captures the common behavior of all the system agents.
b) Define the main and auxiliary methods that implement the agent's basic services.
c) Define the agent actions as methods.
d) Define the agent plans as classes.Specify plan actions as methods of plan classes.
e) Define the agent beliefs as simple strings or classes.
f) The knowledge elements are subclasses of the Belief, Goal and Plan classes.

Step D2. Refine the Interaction aspectual component.

a) Define the interaction infrastructures and corresponding sensors and effectors.
b) Define the agent's internal message format.
c) Define parsers for translating external messages to the internal message format.
d) Create the abstract and concrete aspects to modularize the interaction concern.
e) Refine the sensory interfaces, defining the external objects to be observed.
f) Refine the message reception interfaces, defining the join points where messages should be received.
g) Refine the message sending interfaces, picking out the joint points where messages should be sent from the agent.

Step D3. Refine the Adaptation aspectual component.

a) Create the abstract and concrete aspects to modularize the adaptation concern.
b) Refine the knowledge adaptation interfaces, enumerating the agent's internal events to be observed.
c) Refine the behavior adaptation interfaces, enumerating the agent's internal events to be monitored.

Step D4. Refine the Autonomy aspectual component.

a) Define the reactive, decision and proactive goals for the agent.
b) Create the abstract and concrete aspects to modularize the autonomy concern.
c) Refine the thread management interface, specifying the join points where threads should be started and finalized.
d) Refine the goal creation interface, specifying the events to instantiate goals.
e) Refine the decision-making interfaces for capturing events that trigger agent decisions.
f) Refine the crosscutting interfaces for capturing the events that affect the agent's autonomy degree.

Step D5. Refine the Mobility aspectual component.

a) Create the abstract and concrete aspects to modularize the mobility concern.
b) Refine the crosscutting interfaces for mobile elements, specifying the elements to be moved together with the agent.
c) Refine the crosscutting interfaces for agent travel, picking out the join points that trigger the agent travel and the agent return.

Step D6. Refine the Learning aspectual component.

a) Create the abstract and concrete aspects to modularize the learning concern.
b) Refine the crosscutting interface for information gathering, describing the join points to provide information and trigger the learning process.
c) Refine the crosscutting interfaces for learning knowledge, specifying the attributes and methods with learning-specific knowledge.

Step D7. Refine the Collaboration aspectual component.

a) Create the abstract and concrete aspects to modularize the collaboration concern.
b) Define the collaboration protocols and the corresponding roles.
c) Refine the crosscutting interfaces for enforcing the collaboration protocols.
d) For each role:
    a. Refine the interfaces for the role binding, describing the join points where the role should be bound to the agent.
    b. Refine the interfaces for role-specific knowledge, describing the methods and attributes with role knowledge which should be introduced to the agent.
    c. Refine the role aspects by starting from Step D2.

4.4 Implementation Stage

There are several aspect-oriented programming languages to support the implementation of aspect-oriented agent architectures, such as AspectJ [10] and Hyper/J [42]. AspectJ is the most widely used programming language, which extends the Java programming language. The implementation of aspect-oriented agent architectures in AspectJ is straightforward, since this language supports the definition of inter-type

declarations, pointcuts and advices. However, there are some implementation steps that require some guidance due to AspectJ features and restrictions as summarized below. The reader can find a extensive list of implementation guidelines at [5].

For example, each agent instance must have, in general, its own instance of agenthood or additional aspect. As a consequence, the agent aspects must be instantiated per Agent instance. The current version of AspectJ supports the specification of per-object aspects. We could describe the instantiation of the agent aspects using perthis. However, the use of perthis restricts the scope of the aspect. When one AspectJ aspect is declared to be singleton or static, its scope is the whole system and the aspect can crosscut all system classes. Per-object aspects can only crosscut the object with which it is associated. Since agent concerns crosscut several classes, not only the Agent class or the Role class, the perthis clause cannot be used in this context. As a result, agent aspects are declared as singletons and introduce the methods and attributes to the Agent and Role classes as inter-type declarations.

5 ExpertCommittee: The Case Study

This section introduces a MAS in order to illustrate the application of the guidelines presented in the previous section. This system is a prototype derived from a case study undertaken in the Software Engineering Laboratory at PUC-Rio in Brazil, from herein referred to as EC (ExpertCommittee). EC is an open multi-agent system that supports the management of paper submissions and the reviewing process for a conference. The EC system has been chosen because it is a classical example of agent-based application [43] and it involves all the agenthood and additional properties. This system includes several combinations of agent concerns, which are typical of many existing agent-driven applications.

The EC system encompasses two agent types: user agents and information agents. Each agent type provides different services, but everyone is interactive, adaptive and autonomous. The architecture of each agent type has different agent properties. For simplicity purposes, this section focuses on the description of the user agents. User agents are software assistants that automate time-consuming tasks of paper authors, chairs, PC members and reviewers and coordinate their activities. Fig. 1 shows a partial representation of the object-oriented design of the EC system.

5.1 The Architectural Stage

We describe below the accomplishment of the architectural steps (Section 4.2) to define the architectural design of the user agents. Fig. 4 depicts the architectural view of the EC user agents. The user agents have all the agenthood and additional architectural aspects.

Step A1. The Kernel component of user agents has a normal interface, which makes the agent services available to the environment (A1.a). The agent services can be accessed either by sending an asynchronous request through the communication infrastructure or by directly invoking them. The agents also have a normal interface for accessing and updating the knowledge elements (A1.b). This is a private interface and is not accessible by elements external to the agent.

Step A2. User agents have an Interaction component that implements the three crosscutting interfaces: ISensory, IMessageReception, and IMessageSending. The Sensory interface senses events in environment objects (A2.a). The IMessageReception interface intercepts the messages arrival (A2.b). The IMessageSending interface defines when messages need to be sent from agent plans and actions (A2.c). The Interaction aspect's interfaces crosscut the Kernel component, the Collaboration component, and the Environment component. The Environment component represents the communication platform and the external entities observed or monitored by the agent.

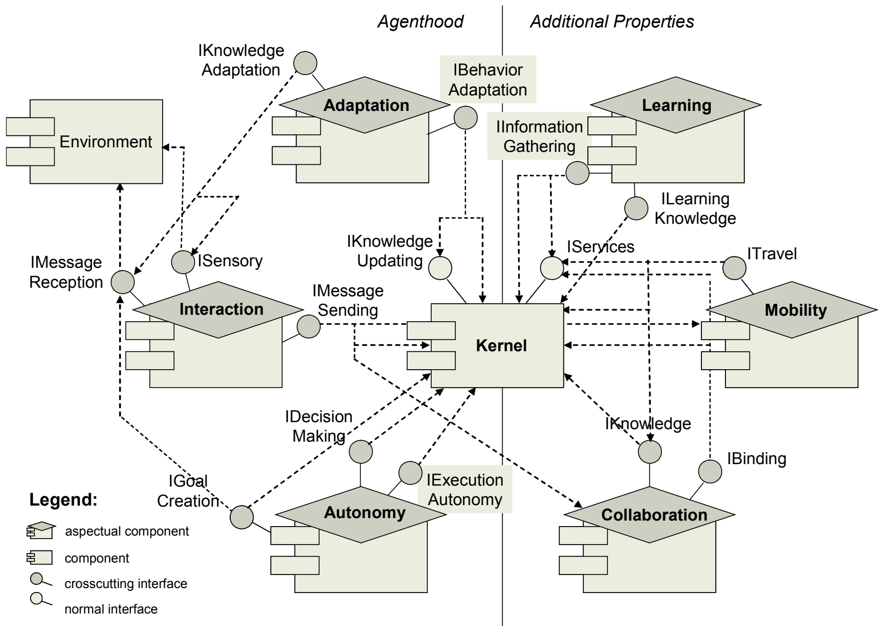


Fig. 4. The Aspect-Oriented Agent Architecture of User Agents

Step A3. The adaptive behavior of user agents involves two kinds of adaptation: knowledge adaptation and behavior adaptation. As a consequence, they have a Adaptation component that realizes the crosscutting interfaces for both of them: IKnowledgeAdaptation and IBehaviorAdaptation. The Adaptation component crosscuts the Interaction component and the Kernel component. It is connected with the former through the IKnowledgeAdaptation interface since knowledge adaptation may be required upon the receipt of external messages (A3.a). The connection with the later is because knowledge adaptation is necessary whenever given internal events happen, such as the change of beliefs. In addition, the Adaptation component crosscuts the Kernel component through the IBehaviorAdaptation interface, since the selection of a new plan is necessary whenever a new goal is set, and the plan execution may have to be canceled due to the change of specific beliefs (A3.b).

Step A4. The Autonomy component conforms to the following crosscutting interfaces: (i) IExecutionAutonomy, which specifies the kernel initialization as the join point to create the agent threads (A4.a), (ii) IGoalCreation, which crosscuts the Interaction

component and the Kernel component because it may be necessary to create goals whenever messages are received and pieces of the agent knowledge are changed (A4.b), and (iii) IDecisionMaking triggers the agent decisions (A4.d). User agents do not have an interface to control the autonomy degree (A4.c).

Step A5. User agents have a Mobility component that conforms to two crosscutting interfaces: IMobileElement and ITravel. IMobileElement specifies all the Kernel elements as mobile elements because at least the agent kernel needs to be moved when the agent departs to a remove environment (A5.a). ITravel crosscuts the Kernel and Collaboration components since the execution of actions and plans triggers agent travels across different hosts as well as the agent return to its home host (A5.b).

Step A6. User agents have a Learning component with two crosscutting interfaces. The first one, IInformationGathering, defines internal events in the Kernel component as the information sources to be observed for learning purposes (A6.a). The second one, ILearningKnowledge, specifies the learning-specific knowledge to be introduced to the agent kernel (A6.b).

Step A7. EC agents have also a Collaboration component that aggregates the roles played by the agents. The interfaces for enforcing collaboration protocols (A7.a) are not represented in Fig. 4 since EC agents do not require this feature. It determines when a given role is bound to the agent. Inner aspectual components represent the agent roles. The Collaboration component is formed by four inner Role components (A7.b), each one for a specific agent role: author, reviewer, PC member, and chair. Each of the Role components implements the IBinding interface and the IKnowledge interface.

5.2 The Detailed Design Stage

Due to space limitations, in this work, we discuss in detail only the steps involving the agenthood concerns (D2-D4). The other steps (D1, D5-D7) are shortly described. A more detailed description is found at [5]. Some figures are used to illustrate the detailed design of the architectural aspects, which crosscut several agent classes and aspects in the EC system. However, for simplification reasons, the figures only present some of these classes and aspects. The others essentially follow the same pattern. The figures represent the crosscut elements in gray.

The aSideML language (Section 4.1) also supports the modeling of the detailed design of aspects. The full view of an aspect provides a detailed description of its elements. An aspect is represented by a rectangle, like classes, with a diamond on its top. The aspect's internal structure declares the internal attributes and methods. Each crosscutting interface is presented using the rectangle symbol with compartments. The first compartment of a crosscutting interface represents inter-type declarations, and the second compartment represents pointcuts and their attached advices. The notation uses a dashed arrow to represent the crosscutting relationship.

Step D1. The Kernel component is refined as a set of classes, which represent the agent itself, and knowledge elements (goals, beliefs, and plans). The Agent class specifies the behavior common to the system's agent types. The UserAgent class extends the Agent class (D1.a), and contains the methods that implement the agent

actions and agent's basic services (D1.b, D1.c). The knowledge elements of user agents are subclasses of the Belief, Goal and Plan (D1.f). Attributes of Agent subclasses can be used to represent simple agent beliefs. Plan actions are methods of Plan subclasses (D1.d). The agent beliefs are attributes of the Agent subclasses (D1.e).

Step D2. User agents interact with the environment using two communication infrastructures: JADE [15] and a blackboard architecture (D2.a). The blackboard is used for internal communication between the agents and the JADE infrastructure is used to interact with agents external to the system. The effectors and sensors associated with these infrastructures are represented by separate class hierarchies (D2.a). The Sensor and Effector subclasses represent sensors and effectors respectively, and cooperate with environment classes. ACL [36] is the used communication language. The agents also use an internal communication language, which is also compliant to the FIPA specification [36] (D2.b). Specific classes are responsible for implementing the parsers (D2.c).

The Interaction architectural component is decomposed in an abstract aspect (D2.d), a concrete aspect (D2.d), and various auxiliary classes. Fig. 5 shows only the aspects, sensors/effectors, and the crosscut elements (in gray); it omits the auxiliary classes. The abstract aspect defines the interaction logic, which is common to all the agent types and roles. It holds the inbox, the outbox, an abstract initialization method, and methods to marshal and unmarshal the messages. This aspect also refines the three crosscutting interfaces defined in the Interaction architectural aspect: ISensory, IMessageSending, and IMessageReception.

The ISensory interface (D2.e) implements the abstract sensing pointcut that declares which methods of the environment classes must be monitored. The sensing_ advice processes the external events and updates the inbox. The sensing pointcut is also declared as abstract since the join points depend on the specific agent types and roles.

The IMessageReception interface (D2.f) introduces the receiveMsg() method to the Agent class in order to enable it to receive messages (inter-type declaration). This interface also defines an incomingMsg pointcut for intercepting executions of the method sense() on the Sensor classes; the goal is to detect the arrival of messages. This pointcut is associated with an after advice responsible for processing the incoming messages and updating the inbox.

The IMessageSending interface (D2.g) extends the Agent class to enable it to send messages, by introducing the sendMsg() method to the Agent class. The interface also defines an outgoingMsg pointcut that specifies the message senders. Note that the outgoingMsg pointcut is abstract (Fig. 5) because the join points depend on the specific agent types and roles. The pointcut is concretized in the Interaction subaspects. This pointcut contains an advice which runs after executions of those join points. The purpose of the advice is to capture the information needed to send the message and update the outbox. Note that the scattering of the interaction concern presented in Fig. 1 is overcome in the aspect-oriented solution (Fig. 5).

The concrete aspect, called UserAgentInteraction, extends the Interaction aspect to define the interaction behavior specific to the user agent. It implements the sensing pointcut by specifying DB components, GUI objects, and business logic components as environment entities to be observed. User agents monitor these elements in order to adapt their knowledge and behavior and learn about the user preferences. This spe-

cific aspect also implements the initialization methods, and the outgoingMsg pointcut that specifies join points in the agent elements from which messages need to be sent to the external world, such as methods of Plan subclasses, Agent subclasses, and Role aspects (D2.g). This pointcut also crosscuts the Mobility aspect because the user agent needs to send notification messages to local agents before moving to a remote environment and after returning to the original environment.

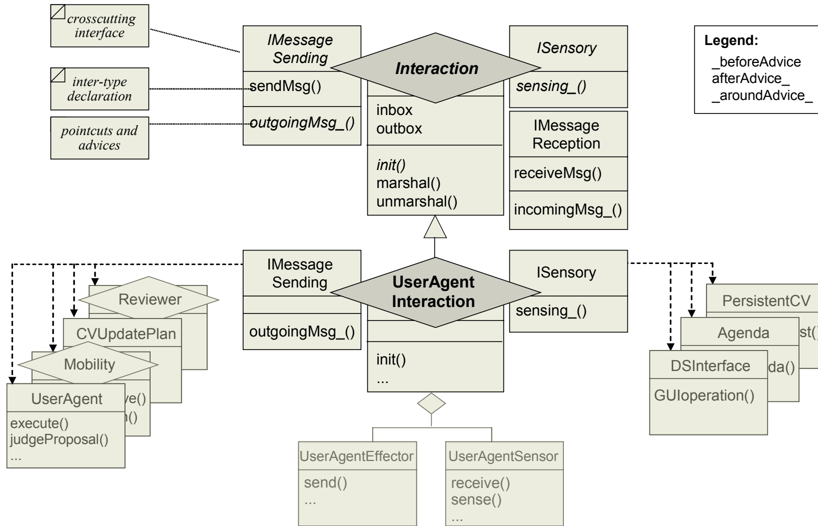


Fig. 5. The Detailed Design of the Interaction Component.

Step D3. The Adaptation architectural component of EC agents is decomposed in an abstract aspect (D3.a), a concrete aspect (D3.a), and auxiliary classes. The abstract aspect defines the generic adaptation protocol: events are sensed, conditions checked, and adapters triggered. The aspect holds the adapter methods and a list of the adapter objects. It contains the advices which either invoke either adapter methods or a specific adapter. The Adaptation aspect is extended by the UserAgentAdaptation aspect to implement the adaptive behavior for the specific context of the user agents.

The abstract aspect also implements the two crosscutting interfaces: IKnowledgeAdaptation (D3.b) and IBehaviorAdaptation (D3.c). They define pointcuts with generic events that always trigger the knowledge and behavior adaptation, independently from the agent type. The agent adaptation occurs in several circumstances: due to external events – for example, message receptions – or due to internal events, belief changes, new goal setting, exceptions thrown during a plan execution, and so forth.

Step D4. The Autonomy architectural component is refined as an abstract aspect (D4.b), a concrete aspect (D4.b), and auxiliary classes. There are various Goal subclasses to define the reactive, decision, and proactive goals of the user agents (D4.a). The DecisionPlan and ProactivePlan subclasses modularize the implementation of more sophisticated decision algorithms and proactive strategies. The abstract Autonomy aspect defines the autonomy behavior common to all the agent types in the EC system. The UserAgentAutonomy aspect extends the Autonomy aspect to implement

the autonomous behavior for the specific context of user agents. This aspect holds the decision and proactive goals, an integer number representing the autonomy degree, initialization methods, and defines the autonomy protocol. It implements three crosscutting interfaces: IExecutionAutonomy, IGoalCreation, and IDecisionMaking. These crosscutting interfaces define how the Autonomy aspect crosscuts different classes and other aspects of software agents.

The IExecutionAutonomy interface (D4.c) defines the pointcuts that specify when control threads are attached to and detached from the Agent instances. It defines the execution of Agent constructors as the join point to start the threads, and the agent destruction (execution of the method kill()) as the join point to finalize the threads. There are after advices associated with these pointcuts in order to invoke the components that implement the Active Object pattern [24].

The IGoalCreation interface (D4.d) specifies join points in the agent classes which events need to be detected to start a goal creation. It contains an advice which runs after executions of actions on agent classes, actions on beliefs classes, action on plan classes, and actions on another aspects associated with the agent (for example, Interaction aspects). Since the Autonomy aspect implements the autonomy protocol, it is associated with the agent, plan or belief classes where changes to their state may trigger a goal creation.

The IDecisionMaking interface (D4.e) specifies the receiveMsg() method on the Interaction aspect (D2) as join point because an agent often needs to decide whether and which reactive goal instance should be created depending on the received messages. After the receiveMsg() method is executed, there is an advice that takes the control on the program execution and instantiate, if necessary, the goal decisions associated with the message type. If the decision is positive, a reactive goal is instantiated. Otherwise, a decision plan sends a message to the sender notifying the agent that the service request will not be performed.

Steps D5-D7. The Mobility, Learning and Collaboration architectural aspects are also decomposed in terms of abstract aspects, concrete aspects, and auxiliary classes. Unlike the agenthood aspects, they are not associated with the Agent class because they are not part of the agenthood. They are associated only with the UserAgent class. For example, the Mobility aspects modularize the following issues: (i) the pointcuts that describe the events which may lead the agent to travel to a remote environment or to go back to the home environment, (ii) the advices responsible for checking the need for the agent roaming and for calling the mobility actions, (iii) the data structures and methods which control the agent itinerary, and (iv) the inter-type declarations that specify which agent elements are mobile and serializable. As a consequence, the agent classes are not intermingled with mobility code, therefore improving their maintainability and reusability. JADE is used as the mobility framework; some auxiliary classes connect the mobility aspects with the JADE framework. Each mobility aspect in the EC system crosscuts about 7 classes and aspects.

Learning aspects encapsulate the entire implementation of the learning concern, including the learning-specific knowledge and the information gathering. Fig. 6 shows that the Learning aspect separates the learning protocol from the kernel and other aspects, such as UserAgent class, Plan classes, and role aspects. The Learning aspects connect the execution points (events) on different agent classes with the corresponding learning components, making it transparent to the agent's basic functionality the

particularities of the learning algorithms in use. These aspects are able to crosscut some agent execution points in order to change their normal execution and invoke the learning components. The execution points include the change of a knowledge element, execution of actions on plans, roles, and agent types, or still some threw exception. Auxiliary classes are used to implement different learning techniques. This learning experience is indirect because the agent will build its knowledge through the results of the negotiations. Machine learning is used to address the knowledge acquisition. Distinct learning techniques are used in the EC system: Temporal Difference Learning (TD-Learning) [41] and Least Mean Squares (LMS) [41]. TD-Learning is used by the reviewer role in order to learn the user preferences in the subjects he/she likes to review. LMS is used by the chair to learn about the reviewer preferences. Note that the scattering of the learning concern presented in Fig. 1 is overcome in the aspect-oriented solution (Fig. 6).

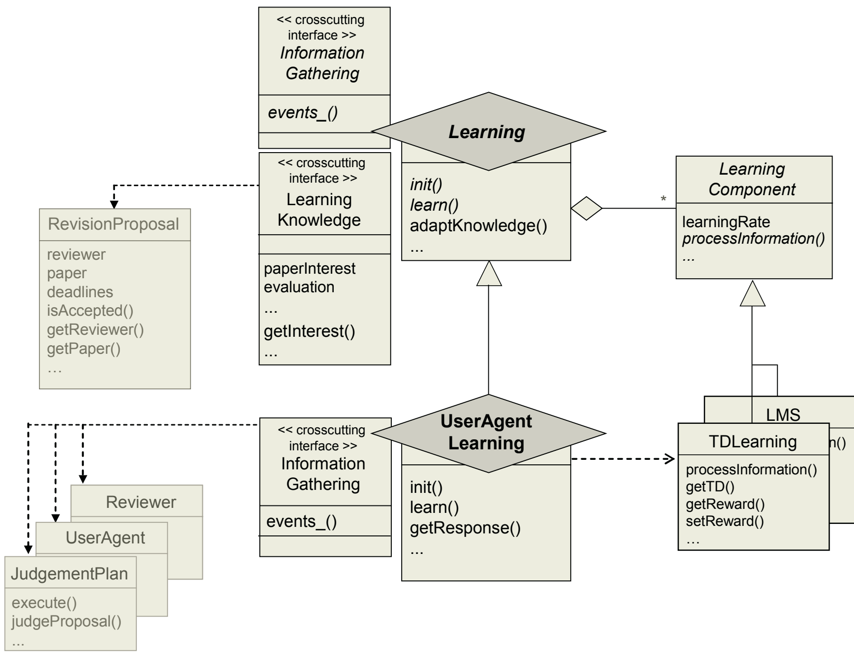


Fig. 6. The Detailed Design of the Learning Component.

5.3 The Implementation Stage

The implementation of the EC system was based on version 1.3 of the AspectJ language [10]. The work [5] presents in detail implementation issues and sample code. The EC implementation also used the JADE framework [15] and a blackboard architecture to support the communication among agents. The integration of the aspect-oriented implementation with those infrastructures was almost straightforward. However, the agent architectures could be also implemented using other aspect-oriented frameworks, such as AspectWerkz and JBoss. Although those frameworks support dynamic weaving, they incorporate constructs similar to AspectJ.

However, some inter-aspect conflicts needed to be solved. For example, the Adaptation and Autonomy aspects have pointcuts defined for the same join point: the executions of the method receiveMsg(). The AspectJ construct declare precedence has been used to specify the order of execution between these aspects. Regarding the interaction concern, there were several join points where messages should be sent to other agents. The join points include methods on plan classes and role aspects. The declaration of all those methods in the pointcut outgoingMsg is time-consuming. In order to facilitate the specification of pointcuts, such methods have been named with the prefix "prepare". The definition of the pointcuts used a simple wildcard prepare* to capture all those methods.

6 Lessons Learned

Three prototypes were built based on our proposed approach and using the AspectJ programming language: (i) a multi-agent system for traffic management [5, 25], (ii) the EC system [5], and (iii) a multi-agent system to manage a development environment for web portals [4, 6, 7]. These systems involved both reactive and cognitive agents with different combinations between the agent concerns. This section presents lessons learned on the design and implementation of aspect-oriented agent architectures, and on empirical assessments [4, 5, 6].

Inseparable Concerns. The Interaction aspects do not modularize the message assembling from different plans or roles; the message needs to be prepared within a method on plan classes or on role aspects because its assembling is very coupled to the role or plan context. One solution would be to separate the message assembling with aspects, but it would result in higher complexity.

Repetitive and Time-Consuming Definitions. All the message senders of the system must be specified in the pointcut inside the Interaction aspect. This might indeed be repetitive and tedious, suggesting that AspectJ should have more powerful metaprogramming constructs. However, this is not an unsolvable problem because code-generation tools can assist MAS engineers in this development step. In addition, we can establish a naming convention and use wildcards supported by most aspect-oriented languages. The implementation of the EC system used naming conventions.

Required Refactoring. In some circumstances, refactoring of the already defined aspects or classes may be needed as the system development evolves. For instance, the realization of the Autonomy architectural aspect requires restructuring of the base code associated with other agent components in order to expose suitable join points. For instance, we need to enforce that each method which asks for the user confirmation (when an agent decision is taken) returns a boolean value. This allows the aspect to capture the user response and control the agent autonomy degree. In addition, we have extracted code from existing methods into a new method to expose a method-level join point. Tools to help with the restructuring would make it easier to introduce aspects into an existing system.

Complex Structure for Simple Agents. Some simple reactive agents do not require thread control, react only to few events, make very simple decisions, and do not have proactive behavior. In this case, the autonomy code tends to be localized in fewer

methods. The use of aspects in this specific situation can increase rather than decrease the agent complexity.

Iterative Process. During our case studies [4, 5, 7, 22, 25], we have tried to "incrementally" deal with agent concerns at the architecture and design stages, following the order prescribed in Section 4. We have found that, as the MASs increases in complexity, the boundary between increments is not as transparent as implied. For example, the design and implementation of the mobility aspects required the creation of new pointcuts in the interaction aspects previously defined. In this way, we mostly had to follow an iterative process rather than an incremental approach in order to implement the aspect-oriented agent architectures.

Empirical Evidence. A systematic evaluation has been carried out to assess the proposed aspect-oriented approach with respect to relevant quantitative criteria [4]. We have compared quantitatively our architectural approach with a mediator-based architecture [4, 5] using a metric-based assessment framework [8]. The tallies of lines of code and number of attributes for the developed MAS in the mediator-based implementation were respectively 12% and 9% higher than in the aspect-oriented code. The aspect-oriented project also produced better results in terms of complexity of operations (6%), component couplings (9%), and component cohesion (3%). The complete description of the data gathered in this experiment can be found in [4].

7 Related Work

Dealing with several agent concerns, such as adaptation and learning, at the phase of architecture definition has been recognized as a serious problem that has not received enough attention [11, 12, 14]. In fact, related work in this area has been scarce, making no attempt in considering agent concerns within the architectural stage. Research in agent-oriented software engineering has concentrated on high-level methodologies and modeling languages [17]. Our previous work [6] dealt with crosscutting agent concerns, but it was a initial version of our approach and was focused on the detailed design and implementation levels.

Section 3 presented the existing architectural approaches for the separation of agent concerns. Software architects want to separate the application concerns in separate components, but the existing architectural styles are not able to address this separation in multi-agent systems. Our proposed agent architecture is different from a mediator-based architecture because the composition of agent concerns is not centralized in a single component, the mediator. Each architectural aspect specifies how it affects the other architectural components. The proposed architecture is not a reflective architecture since architectural aspects, unlike meta-objects, are not limited to be attached to a single object. In addition, architectural aspects can change the interface of other components by introducing fields and methods to them. Finally, the aspect-oriented agent architecture is also different from layered agent architectures defined in Kendall's approach [26]. The architectural aspects are not structured as layers; each architectural component can be associated with more than two components.

The proposed approach describes a set of architectural decisions, which contribute to improve the maintainability of MASs. The achieved segregation limits significantly

the impact of a change since the architectural components modularize the crosscutting agent concerns. Unlike the use of mediator-based agent architectures, the use of aspect-oriented architectures support the functional encapsulation of the agent's basic functionality since the Kernel component is not intermingled with agent properties. The crosscutting interfaces allow the addition of agenthood and additional properties to the basic functionality in a way that is not intrusive. As a consequence, the architectures of existing objects can be transformed into agent architectures without any changes to their methods.

The aspect-oriented architecture also improves the chances for reuse of the agent components. Applications that adopt this architecture can reuse and refine the architectural components in a more modular way since the crosscutting agent concerns are encapsulated in aspects. The agent concerns are not scattered and tangled up with each other. The improved separation of concerns facilitates also the construction of heterogeneous agent architectures. Each architectural component is oblivious in how it is modified by agent aspects. There is no reference in the Kernel component to the agent aspects. As a consequence, it is easier add or remove aspects from the agent architecture.

8 Conclusions and Ongoing Work

This paper presented an aspect-oriented stepwise approach meant to be simple enough to be used in the development of reusable and maintainable agent architectures in different kinds of agent-oriented systems. The approach follows an aspect-oriented agent architecture [5], which is a high-level description of the agents' internal structure in terms of architectural aspects and their relationships. The proposed aspect-oriented approach supports: (i) the modularization of crosscutting agent concerns from the architectural definition to the system implementation, (ii) the flexible integration of the agent concerns, and (iii) the independence of programming languages or MAS implementation frameworks. Since the aspect-oriented method is independent of programming language or implementation framework, a wide range of application developers can employ it.

The use of the aspect-oriented architecture can minimize the complexity caused by the crosscutting nature of agent properties. It proposes the use of aspects to provide a clear separation of concerns between the agent's basic functionality and the crosscutting agent properties. Moreover, aspect-oriented architectures allow that the agent properties to be incorporated into an object-oriented system when the developers want to transform their predefined objects into agents. The incorporation of agent properties can be made by attaching the corresponding agent aspects to the existing non-agent objects. However, some refactoring of the predefined objects may be required to expose the suitable join points, as discussed in Section 6.1.

We have also worked on the definition of a pattern language to support the detailed design of aspect-oriented agent architectures [5]. Each pattern in the language provides an aspect-oriented design solution for a specific crosscutting agent concern, such as learning [22] and mobility [18]. As future work, we are planning to study whether crosscutting agent concerns need to be managed at the requirements-level and if so how to support this management.

Acknowledgements

This work has been partially supported by CNPq under grants No. 141457/2000-7 and No. 381724/2004-2 for Alessandro, grant No. 140252/2003-7 for Uirá, and by FAPERJ under grant No. E-26/150.699/2002 for Alessandro. The authors are also supported by the PRONEX Project under grant 7697102900, and by ESSMA under grant 552068/2002-0.

References

1. F. Buschmann et al. Pattern-Oriented Software Architecture: A System of Patterns. John Wiley Sons, 1996.
2. C. Chavez. A Model-Driven Approach to Aspect-Oriented Design. PhD Thesis, Computer Science Department, PUC-Rio, April 2004, Rio de Janeiro, Brazil.
3. A. Garcia, M. Cortés, C. Lucena. A Web Environment for the Development of E-Commerce Portals. Proceedings of the IRMA'01, Toronto, May 2001.
4. A. Garcia et al. Separation of Concerns in Multi-Agent Systems: An Empirical Study. In: "Software Engineering for Multi-Agent Systems II", Springer, LNCS 2940, April 2004.
5. A. Garcia. From Objects to Agents: An Aspect-Oriented Approach. PhD Thesis, Computer Science Department, PUC-Rio, April 2004, Rio de Janeiro, Brazil.
6. A. Garcia, C. Lucena, D. Cowan. Agents in Object-Oriented Software Engineering. Software: Practice and Experience, Volume 34, Issue 5, April 2004, pp. 489-521.
7. A. Garcia et al. Engineering Multi-Agent Systems with Aspects and Patterns. Journal of the Brazilian Computer Society, Number 1, Volume 8, July 2002, pp. 57-72.
8. C. Sant'anna, A. Garcia, C. Chavez, C. Lucena, A. Staa. On the Reuse and Maintenance of Aspect-Oriented Software: An Assessment Framework. Proc. 17th Brazilian Symposium on Software Engineering (SBES'03), Manaus, Brazil, October 2003.
9. G.Kiczales, et al. Aspect-Oriented Programming. Proc.ECOOP'97, LNCS 1241, June 1997.
10. G. Kiczales et al. Getting Started with AspectJ. CACM, October 2001.
11. A.Pace et al. Architecting the Design of Multi-Agent Organizations with Proto-Frameworks. In: "Software Engineering for MASs II", LNCS 2940, Feb 2004, pp. 75-92.
12. A. Pace et al. Assisting the Development of Aspect-based MAS using the SmartWeaver Approach. In: "Software Engineering for Large-Scale MASs", LNCS 2603, March 2003.
13. V. Silva et al. "Taming Agents and Objects in Software Engineering". In: "Software Engineering for Large-Scale Multi-Agent Systems", Springer, LNCS 2603, March 2003.
14. N. Ubayashi, T. Tamai. Separation of Concerns in Mobile Agent Applications. Proc. of the 3rd Conference Reflection 2001, LNCS 2192, Kyoto, September 2001, pp. 89-109.
15. F. Bellifemine et al. JADE: A FIPA-Compliant Agent Framework. Proc. of the Practical Applications of Intelligent Agents and Multi-Agents, pp. 97-108, April 1999.
16. A. Fuggetta, G. Picco, C. Vigna. Understanding Code Mobility. IEEE Transactions on Software Engineering, vol.24, No.5, pp.342-361, 1998.
17. C. Iglesias, et al. A Survey of Agent-Oriented Methodologies. Proceedings of the ATAL-98, Paris, France, July 1998, pp. 317-330.
18. A. Garcia et al. The Mobility Aspect Pattern. Proc. of the 4th Latin-American Conference on Pattern Languages of Programming, SugarLoafPLoP'04. August, 2004, Fortaleza, Brazil.
19. A. Amandi, A. Price. Building Object-Agents from a Software Meta-Architecture. In: Advances in Artificial Intelligence, LNAI, vol. 1515, Springer-Verlag, 1998.
20. E. Gamma et al. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, 1995.

21. D. Camacho. Coordination of Planning Agents to Solve Problems in the Web. AI Communications, IOS Press, Vol. 16 (4), November, 2003, pp. 309-311.
22. A. Garcia et al. The Learning Aspect Pattern. Proc. of the 11th Conference on Pattern Languages of Programs (PLoP2004), September 2004, Monticello, USA.
23. E. Pulvermüller, A. Speck, A. Rashid. Implementing collaboration-based Designs using Aspect-Oriented Programming. Proc. of TOOLS-USA, 2000, p. 95 - 104, Jul 2000.
24. R. Lavender, D. Schmidt. Active Object: an Object Behavioral Pattern for Concurrent Programming. In: “Pattern Languages of Program Design”, Addison-Wesley, 1996.
25. A. Costa. An Aspect-Oriented Software Architecture for Traffic Simulators. Master’s Dissertation, University of Sao Paulo, October 2003. (In Portuguese)
26. E. Kendall et al. A Framework for Agent Systems. Implementing Application Frameworks – OO Frameworks at Work, M. Fayad et al. (ed). John Wiley & Sons: 1999.
27. E. Kendall. Role Model Designs and Implementations with Aspect-oriented Programming. Proceedings of OOPSLA’99, ACM Press, 1999, pp. 353-369.
28. M. D'Hondt, K. Gybels, V. Jonckers. Seamless Integration of Rule-Based Knowledge and Object-Oriented Functionality with Linguistic Symbiosis. Proceedings of the 19th Annual ACM Symposium on Applied Computing (SAC 2004), Nicosia, Cyprus, March 2004.
29. E. Dijkstra. A Discipline of Programming. Prentice Hall, Englewood Cliffs, NJ, 1976.
30. N. Jennings. Agent-Oriented Software Engineering. Proc. of the 12th Intl. Conference on Industrial and Engineering Applications of Artificial Intelligence, 1999, pp. 4-10.
31. M. Huhns, M. Singh (Eds.). Agents and Multiagent Systems: Themes, Approaches, and Challenges. Readings in Agents, Chapter 1, Morgan Kaufmann Publishers, USA, pp. 1-23.
32. D. Rasmus. Rethinking Smart Objects: Building Artificial Intelligence with Objects. Cambridge University Press, New York, 1999.
33. J. Briot, L. Gasser. Agents and Concurrent Objects. IEEE Concurrency, Special Issue on Actors and Agents, 1998.
34. A. Rao, M. Georgeff. BDI Agents: From Theory to Practice. Proceedings of the 1st Intl. Conference on Multi-Agent Systems (ICMAS-95), San Francisco, 1995; 312-319.
35. Shoham, Y. Agent-Oriented Programming. Artificial Intelligence, 60(1):51-92, Mar 1993.
36. FIPA, Agent Communication Technical Committee. Agent Communication Language - FIPA'99 Draft Specification, 1999. http://www.fipa.org.
37. S. Splunter, N. Wijngaards, F. Brazier. Structuring Agents for Adaptation. In: E. Alonso et al (Eds), Adaptive Agents and Multi-Agent Systems, LNAI, Vol. 2636, 2003, pp. 174-186.
38. S.Russell, P.Norvig. Artificial Intelligence: A Modern Approach.Prentice Hall, 2 ed, 2002.
39. T. Norman, D. Long. Goal Creation in Motivated Agents. In: Wooldridge, Jennings (Eds.), Intelligent Agents: Theories, Architectures, and Languages, LNAI 890: Springer, 1995.
40. B. Ekdahl. How Autonomous is an Autonomous Agent? Proc. of the 5th Conference on Systemic, Cybernetics and Informatics (SCI 2001), July 22-25, 2001, Orlando, USA.
41. T. Mitchell. Machine Learning. McGraw Hill, New York, 1997.
42. P. Tarr, H. Ossher. Hyper/J User Manual, 2000. www.alphaworks.ibm.com/tech/hyperj
43. F.Zambonelli,N.Jennings,M.Wooldridge. Organizational Abstractions for the Analysis and Design of Multi-agent Systems. In:“Agent-Oriented Software Engineering”, Springer, 2001.
44. Z. Guessoum, J. Briot. From Active Objects to Autonomous Agents. IEEE Concurrency, Special Series on Actors and Agents, Vol. 7, N. 3, 1999, pp. 68-76.

CAMLE: A Caste-Centric Agent-Oriented Modelling Language and Environment

Lijun Shan[1] and Hong Zhu[2]

[1] Department of Computer Science, National University of Defence Technology
Changsha, 410073, P.R. China
lijunshancn@yahoo.com
[2] Department of Computing, Oxford Brookes University, Oxford OX33 1HX, UK
hzhu@brookes.ac.uk

Abstract. This paper presents an agent-oriented modelling language and environment CAMLE. It is based on the conceptual model of multi-agent systems (MAS) proposed and formally defined in the formal specification language SLABS. It is caste-centric because the notion of caste plays the central role in its methodology. Caste is the classifier of agents in our language. It allows multiple and dynamic classifications of agents. It serves as the template of agents and can be used to model a wide variety of MAS concepts, such as roles, agent societies, etc. The language supports modelling MAS at both macro-level for the global properties and behaviours of the system and micro-level for properties and behaviours of the agents. The environment provides tools for constructing graphic MAS models in CAMLE, automatically checking consistency between various views and models at different levels of abstraction, and automatically transforming models into formal specifications in SLABS. The uses of the CAMLE modelling language and environment are illustrated by an example.

1 Introduction

One of the key factors that contribute to the progress in software engineering over the past two decades is the development of increasingly powerful and natural high-level abstractions with which complex systems are modelled, analysed and developed. In recent years, it becomes widely recognized that agents represent an advance in this direction that can unify data abstraction and operation abstraction. A number of agent-oriented software development methodologies have been proposed in the literature; see e.g. [1]. These proposals vary in how to describe agent and MAS at a higher abstraction level as well as how to obtain such a description. For example, Gaia [2] provides software engineers with the organization-oriented abstraction in which software systems are conceived as organized society and agents are seen as role players. Tropos [3] emphasizes the uses of notions related to mental states during all software development phases. The notions like belief, intention, plan, goals, etc., represent the abstraction of agent's state and capability.

Our work originates from formally specifying agent behaviour as responses to environment scenarios [4], developed into a formal specification language SLABS for engineering agent-based systems [5], and applied to a number of examples of MAS [6, 7]. In [8] and [9], a diagrammatic notation for modelling agent behaviours and

R. Choren et al. (Eds.): SELMAS 2004, LNCS 3390, pp. 144–161, 2005.
© Springer-Verlag Berlin Heidelberg 2005

collaborations was respectively developed. This paper proposes a methodology of agent-oriented software engineering called CAMLE, which stands for Caste-centric Agent-oriented Modelling Language and Environment. Caste is the classifier of agents in our modelling and specification languages. It allows multiple classifications (i.e. an agent can belong to more than one caste) and dynamic classifications (i.e. an agent can change its caste membership at run time), as well as multiple inheritances among castes. It can be used to model a wide variety of MAS concepts, such as roles, agent societies, behaviour normality, etc. It provides the modularity language facility and serves as the template of agents in the design and implementation of MAS. The notion of caste plays a central role in the methodology. We consider behaviour rules as the basic abstraction for agent's behaviour while leaving out mental state notions such as belief and goal that are used in some other agent-oriented software researches, though such notions can be represented in our framework. Behaviour rules incorporating agent's perception to its environment represent the autonomy of agent's behaviour. With the CAMLE language, a software system can be modelled from three perspectives. The supporting tools help users to construct MAS models in graphical notations, to check the consistency between models from various views and at different abstraction levels, and automatically translate the graphic models into formal specifications.

The remainder of this paper is organized as follows. Section 2 reviews the underlying conceptual model. Section 3 presents the modelling language. Section 4 briefly reports the modelling tools. Section 5 concludes the paper with discussions on related work and directions for future work.

2 Conceptual Model

The conceptual model of MAS underlying our methodology is the same as that of the language SLABS [4, 5], which is a formal specification language designed for engineering MAS. It can be characterized by a set of pseudo-equations. Pseudo-equation (1) states that agents are defined as real-time active computational entities that encapsulate data, operations and behaviours, and situate in their designated environments.

$$\text{Agent}=\langle\text{Data, Operations, Behaviour}\rangle_{\text{Environment}} \quad (1)$$

Here, data represent an agent's state. Operations are the actions that the agent can take. Behaviour is described by a set of rules that determine how the agent behaves including when and how to take actions and change state in the context of its designated environment. By encapsulation, we mean that an agent's state can only be changed by the agent itself, and the agent can decide 'when to go' and 'whether to say no' according to an explicitly specified set of behaviour rules. Therefore, there are two fundamental differences between objects and agents in our conceptual model. First, objects do not contain any explicitly programmed behaviour rule. Second, objects are open to all computation entities to call its public methods without any distinction of them.

In our conceptual model, the classifier of agents is called caste. Castes classify agents into various castes similar to that data types classify data into types, and classes classify objects into classes. However, different from the notion of class in object orientation, caste allows dynamic classification, i.e. an agent can change its

caste membership (called casteship in the sequel) at run time. It also allows multiple classifications, i.e. an agent can belong to more than one caste at the same time. As all classifiers, inheritance relations can also be specified between castes. As a consequence of multiple classifications, a caste can inherit more than one caste. As a modularity language facility, a caste serves as a template that describes the structure and behaviour properties of agents in the caste, and as the basic organizational units in the design and implementation of MAS. Pseudo-equation (2) states that a caste is a set of agents that have the same structural and behavioural characteristics at any time moment t in the execution of the system. The structure of caste descriptions in SLABS is shown in Fig.1.

$$\text{Caste}_t = \{\text{agents} \mid \text{structure \& behaviour properties}\} \quad (2)$$

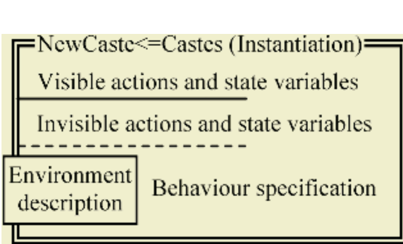


Fig. 1. Caste descriptions in SLABS

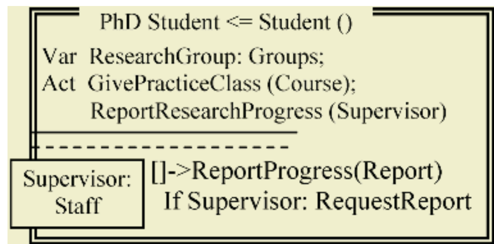


Fig. 2. An example of caste: PhD Student

For example, when modelling a university as an information system, each of the people in the university can be modelled as an agent. They can be grouped into a number of castes, such as the caste of students, the caste of faculty members and the caste of secretaries. The students can be further classified into undergraduates, graduates and Ph. D students. Fig. 2 is an example of the caste definition of Ph. D student.

In the real world, an undergraduate student can become a postgraduate student or alumni after graduation. To model this, the agents in an information system dynamically change their membership to castes. The weakness of static object-class relationship in current mainstream object-oriented programming has been widely recognized. Proposals have been advanced, for example, to allow objects' dynamic reclassification [10]. In [11], we suggested that agents' ability to dynamically change its roles is represented by dynamic casteship. In our model, dynamic casteship is an integral part of agents' behaviour capability. Agents can have behaviour rules that allow them to change their castes at run-time autonomously. To change its casteship, an agent takes an action to join a caste or retreat from a caste at run time. Therefore, which agents are in a caste depends on time even if agents can be persistent, hence the subscript of t in pseudo-question (2). We believe that this feature allows users to model the real world MAS naturally and to maximize the flexibility and power of agent technology.

In the research on agent-oriented methodologies, a number of notions have been proposed in the literature to model agent-based systems, which include role, agent society, organisation, normative behaviour, etc. The notion and language facility of caste can be used to represent these concepts as discussed in [6]. For example, the set of agents that play a specific role can be defined by a caste. The agents of a particular society or community that obey a specific set of normative behaviour rules and share

a set of resources can also be defined as a caste. However, the notions of societies and role etc. are too specific and restrictive to be used as a language facility. For example, all the people who speak a particular language, say Chinese, can be defined as a caste, but it would be unnatural to consider them as playing any specific role. Similarly, a set of software agents that follow a particular communication protocol can be defined by a caste, but it would be unnatural to model them by a role. The concept of society or community has a strong sense of membership. A person who speaks English does not necessarily belong to the society of English people. A society may also consists of agents playing different roles and obey different behaviour rules. Therefore, the concept of society is not suitable to be used as a language facility of code template.

The notion of role has been widely used to characterize agents' behaviour and interaction in agent-oriented methodologies, especially in the analysis and specification stage [13]. However, role is often used intuitively in system analysis. They are transformed into agent properties at design stage and eventually disappear in programming stage or represented indirectly as objects and classes. In contrast, caste not only overcomes the limitations and weakness of the informal notions of roles and societies at analysis and specification stage, but, as a language facility, can also be directly implemented in a programming language such as SLABSp [12].

Equation (3) states that in our model a MAS consists of a set of agents but nothing else. Our definition of agent implies that object is a special case of agent in the sense that it has a fixed rule of behaviour, i.e. "executes the corresponding method when receives a message".

$$\text{MAS} = \{\text{Agent}_n\},\ n \in \text{Integer} \tag{3}$$

Consequently, the environment of an agent in a MAS at time t is a subset of the agents, where some agents in the system may not be visible from the agent's point of view, as illustrated in pseudo-equation (4). Notice that, our use of the term 'visibility' is different from the concept of scope. In particular, from agent A's point of view, agent B is visible means that agent A can observe and perceive the visible actions taken by agent B or obtain the value of agent B's visible part of state at run time.

$$\text{Environment}_t\ (\text{Agent, MAS}) \subseteq \text{MAS} - \{\text{Agent}\} \tag{4}$$

Here, we take a 'designated environment' approach, i.e. the environment of an agent is specified when an agent is designed. The environment description of an agent or a caste defines what kinds of agents are visible. For example, it can be that the agents in a particular caste are visible. Note that a designated environment is neither closed, nor fixed, nor totally open. Since an agent can change its casteship, its environment may change dynamically. For example, an agent's environment changes when it joins a caste and hence the agents in the caste's environment become visible. The environment also changes when other agents join the caste in the agent's environment. Therefore, the set of agents in the environment of an agent depends on time, hence, the subscription t in pseudo-question (4).

The communication mechanism in our model is that an agent's actions and states are divided into the visible ones and internal ones. Agents communicate with each other by taking visible actions and changing visible state variables, and by observing other agents' visible actions and visible states, as shown in pseudo-equation (5). An agent taking a visible action can be understood as generating an event that can be perceived by other agents in the system, while an agent taking an internal action

means it generates an event that can only be perceived by its components. Similarly, the value of an agent's visible state can be obtained by other agents, while the value of the internal state can only be obtained by its components.

$$A \rightarrow B = A.Action \;\&\; B.Observation \qquad (5)$$

This communication mechanism is different from message passing between objects where each message invokes a corresponding method of the object that receives the message. In our model, agents are active computational entities that execute concurrently. They are not invoked by messages. Instead, each agent observes the events happened in its environment and takes actions according to its behaviour rules. How an agent handles an event that it perceives is solely determined by the agent itself because agents are autonomous. In general, the agent that produces an event may not know which agent in the system will respond to the event or how the event will be handled. Therefore, the agent does not expect any agent to participate in the generation of an event. It may not even wait for the event to be handled to progress its own computation task. In this sense, the communication mechanism can be considered as asynchronous and non-blocking. Of course, this mechanism does not define the communication protocol and agent communication language. These issues should be addressed in the design and implementation of specific multi-agent system, rather than predefined by the modelling language or meta-model.

3 Modelling Language

CAMLE employs the multiple view principle. A MAS model contains three types of models: caste models, collaboration models and behaviour models. Each model consists of one or more diagrams.

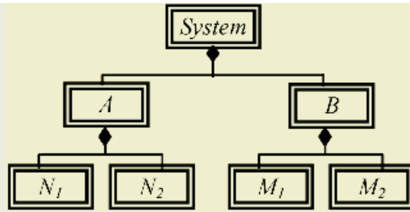
The caste model specifies the castes of the system and the relationships between them. A caste is a compound caste if its agents are composed of a number of other agents; otherwise, it is atomic. For example, as shown in Fig 3 (a), the System is directly composed of agents of caste A and B. Each of them can be further decomposed into smaller components N_1 and N_2, and M_1 and M_2, respectively. For each compound caste, such as the System, A and B, a collaboration model and a behaviour model are constructed. Atomic castes only have behaviour models because they have no components thus no internal collaboration.

The overall structure of a system's collaboration models and behaviour models can be viewed as a hierarchy, which is isomorphic to the whole-part relations described in the caste model; see e.g. Fig 3 (b).

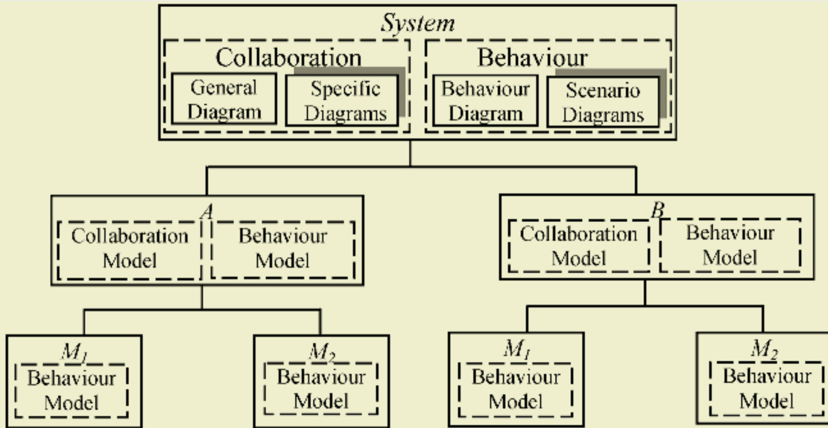
The following subsections describe each type of model and discuss their uses in agent-oriented software development, respectively; see [8, 9] for more details. Subsection 3. 4 discusses the consistency between various kinds of models.

3.1 Caste Model

We view an information system as an organization that consists of a collection of agents that stand in certain relationships to one another by being a member of certain groups and playing certain roles, i.e. in certain castes. They interact with each other by observing their environments and taking visible actions as responses to the environment scenarios. The behaviour of an individual agent in a system is determined by



(a) Example of Caste Model with Whole-Part Relations



(b) Collaboration Models and Behaviour models

Fig. 3. Overall Structure of CAMLE Models

the 'roles' it is playing. An individual agent can change its role in the system. However, the set of roles and the assignments of responsibilities and tasks to roles are usually quite stable [13]. Such an organizational structure of information systems is captured in our caste model.

A caste diagram identifies the castes in a system, indicates the inheritance, aggregation and migration relationships between them. Fig 4 shows the notation of caste diagrams and illustrates it with the university example introduced in section 2.

The inheritance relationship between castes defines sub-groups of the agents that have special responsibilities and hence additional capabilities and behaviours. Migration relations specify how agents in the castes can change their casteships. There are two kinds of migration relationships: migrate and participate. A migrate relation from caste A to B means that an agent of caste A can retreat from caste A and join caste B. A participate relation from caste A to B means that an agent of caste A can join caste B while retaining its casteship of A. For example, in Fig 4, an undergraduate student may become a postgraduate after graduation. A postgraduate student may become a PhD student after graduation or become a faculty member. Each student becomes a member of the alumni of the university after leaving the university. A faculty member can become a part time PhD student while remaining employed as a faculty member. From this model, we can infer that an individual can be both a student and a faculty member at the same time if and only if he/she is a PhD student.

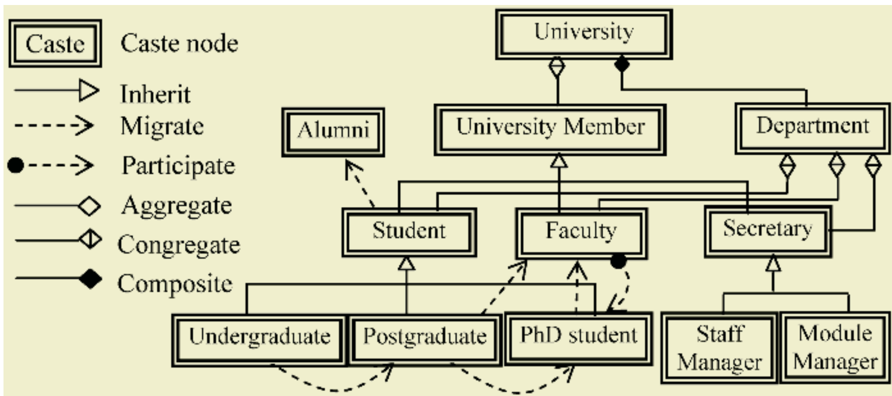


Fig. 4. Caste diagram: notations and example

An agent may contain a number of components that are also agents. The former is called compound agent of the latter. In such a case, there exists a whole-part relationship between the compound agent and the components. We identify three types of whole-part relationships between agents according to the ways a component agent is bound to the compound agent and the ways the compound agent controls its components. The strongest binding between a compound agent and its components is *composition* in which the compound agent is responsible for creation and destruction of its components. If the compound agent no longer exists, the components will not exist. The weakest binding is *aggregation*, in which the compound and the component are independent to each other, so that the component agent will not be affected for both its existence and casteships when the compound agent is destroyed. The third whole-part relation is called *congregation*. It means if the compound agent is destroyed, the component agents will still exist, but they will lose the casteship of the component caste.

The composition and aggregation relation is similar to the composition and aggregation in UML, respectively. However, congregation is a novel concept in modelling languages introduced in by CAMLE. There is no similar counterpart in object oriented modelling languages, such as UML. It has not been recognized in the research on object-oriented modelling of whole-part relations [14]. We believe that it is important for agent-oriented modelling because of agents' basic feature of dynamic casteship. For example, as shown in Fig 4, a university consists of a number of individuals as its members. If the university is destroyed, the individuals should still exist. However, they will lose the membership as the university member. Therefore, the whole-part relationship between University and University Member is a congregation relation. This relationship is different from the relationship between a university and its departments. Departments are components of a university. If a university is destroyed, its departments will no long exist. The whole-part relationship between Department and University is therefore a composition relation.

The semantics of the whole-part relations at modelling level given above has a number of implications on the operations on agents at implementation level , especially the creation and destroy of agents. For example, a composition relation implies that a component agent can be destroyed when the compound agent is destroyed.

In contrast, a component agent must be kept intact even if the compound agent is destroyed if the whole-part relation is aggregate. In object-oriented systems, a component object can be destroyed or garbage collected if there is no more reference to the component object. Therefore, to support garbage collection, a reference count to a component object should be maintained and the reference count must be decreased if the compound object is destroyed. However, in agent-oriented systems, because agents are active computation entities, an agent cannot be destroyed unless explicitly instructed by the user even if it is not a component of any compound agent. For congregation relation, when a compound agent is destroyed, the component agents should not be destroyed, but their casteship must be changed so that they are no longer members of the component castes of the compound agent. It is an open question whether or not an agent should be destroyed if it no longer belongs to any caste.

3.2 Collaboration Model

While caste model defines the static architecture of MAS, collaboration model implicitly defines the dynamic aspect of the MAS organization by capturing the collaboration dependencies and relationships between the agents.

Agents in a MAS collaborate with each other through communication, which is essential to fulfil the system's functionality. Such interactions between agents are captured and represented in a collaboration model. In CAMLE, a collaboration model is associated to each caste and consists of a set of collaboration diagrams.

A collaboration diagram specifies the interaction between the agents in the system or in a compound agent. Fig. 5 gives the notations.

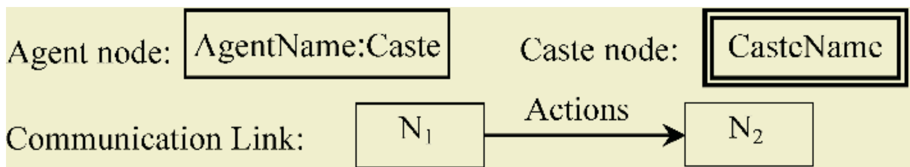


Fig. 5. Notation of Collaboration Diagram

There are two types of nodes in a collaboration diagram. An agent node represents a specific agent. A caste node represents any agent in a caste. An arrow from node A to node B represents that the visible behaviour of agent A is observed by agent B. Therefore, agent A influences agent B. When agent B is particularly interested in certain activities of agent A, the activities can also be annotated to the arrow from A to B. Although this model looks similar to collaboration diagrams in UML, there are significant differences in the semantics. In OO paradigm, what is annotated on the arrow from A to B is a method of B. It represents a method call from object A to object B, and consequently, object B must execute the method. In contrast, in CAMLE the action annotated on an arrow from A to B is a visible action of A. Moreover, agent B is not necessarily to respond to agent A's action. The distinction indicates the shift of modelling focus from controls represented as method calls in OO paradigm to collaborations represented as signalling and observation of visible actions. It fits well with the autonomous nature of agents.

3.2.1 Scenarios of Collaboration. One of the complications in the development of collaboration models is to deal with agents' various behaviours in different scenarios. By scenario, we mean a typical situation of the operation of the system. In different scenarios, agents may pass around different sequences of messages and may communicate with different agents. Therefore, it is better to describe them separately. The collaboration model supports the separation of scenarios by including a set of collaboration diagrams. Each diagram represents one scenario. In such a scenario specific collaboration diagram, actions annotated on arrows can be numbered by their temporal sequence. Fig. 6 below gives an example of scenario-specific collaboration diagram. It describes the collaborations of an undergraduate student with his/her personal tutor, the faculty members who give lectures and the PhD students who are practical class tutors.

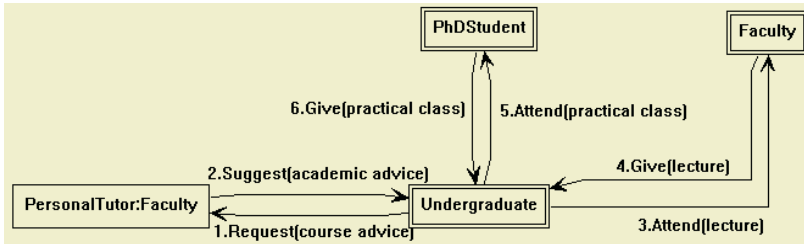


Fig. 6. An example of Scenario-Specific Collaboration Diagram

In addition to such specific diagrams, a general collaboration diagram is also associated to the caste to give an overall picture of the communication between all the component agents by describing all visible actions an agent may take and all possible observers of the actions. Fig. 7 describes the communications within a department between various agents.

3.2.2 Refinement of Collaboration Models. The modelling language supports modelling complex systems at various levels of abstraction, and to refine from high-level models of coarse granularity to more detailed fine granularity models. At the top level, a system can be viewed as an agent that interacts with users and/or other systems in its external environment. This system can be decomposed into a number of subsystems interacting with each other. A sub-system can also be viewed as an agent and further decomposed. As analysis deepens, a hierarchical structure of the system emerges. In this way, the compound agent has its functionality decomposed through the decomposition of its structure. Such a refinement can be carried on until the problem is specified adequately in detail. Thus, a collaboration model at system level that specifies the boundaries of the application can be eventually refined into a hierarchy of collaboration models at various abstraction levels. Of course, the hierarchical structure of collaboration diagrams can also be used for bottom-up design and composition of existing components to form a system.

Fig. 8 gives an example of general collaboration diagram that refines the caste Dept Office. In this diagram, the agents in the castes of Student and Faculty as well as a specific agent called Dept Head in the caste of Faculty form the environment of the caste Dept Office. Therefore, they are visible for the component agents of the caste.

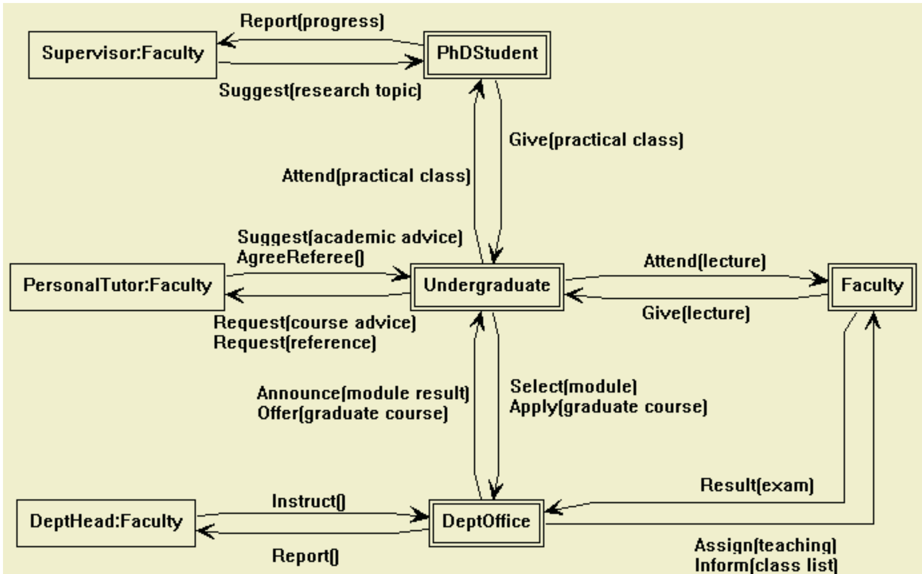


Fig. 7. An example of general collaboration diagram

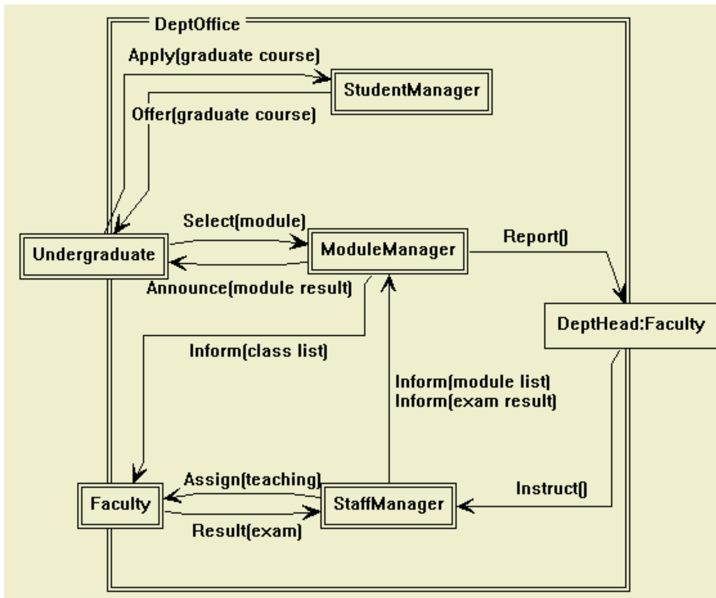


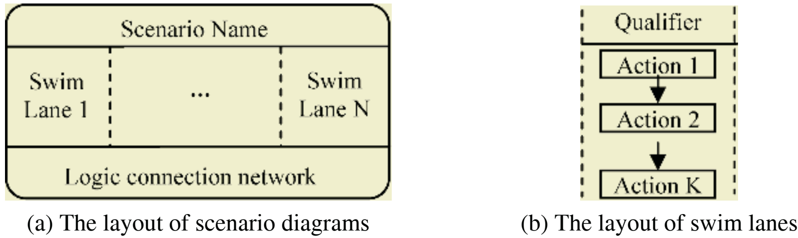
Fig. 8. An example of general collaboration diagram that refines a caste

3.3 Behaviour Model

While caste and collaboration models describe MAS at the macro-level from the perspective of an external observer, behaviour model adopts the internal or first-person

view of each agent. It describes an agent's dynamic behaviour in terms of how it acts in certain scenarios of the environment. A behaviour model consists of two kinds of diagrams: scenario diagrams and behaviour diagrams.

3.3.1 Scenario Diagrams. From an agent's point of view, the situation of its environment is characterized by what is observable by the agent. In other words, a scenario is defined by the sequences of visible actions taken by the agents in its environment. Scenario diagrams identify and describe the typical situations that the agent must respond to. Fig. 9 shows the layout of scenario diagrams. Fig. 10 gives the notations for specifying visible events and their temporal ordering in scenario diagrams, as well as logic connective for the combination of situations.



(a) The layout of scenario diagrams (b) The layout of swim lanes

Fig. 9. Format of Scenario Diagram

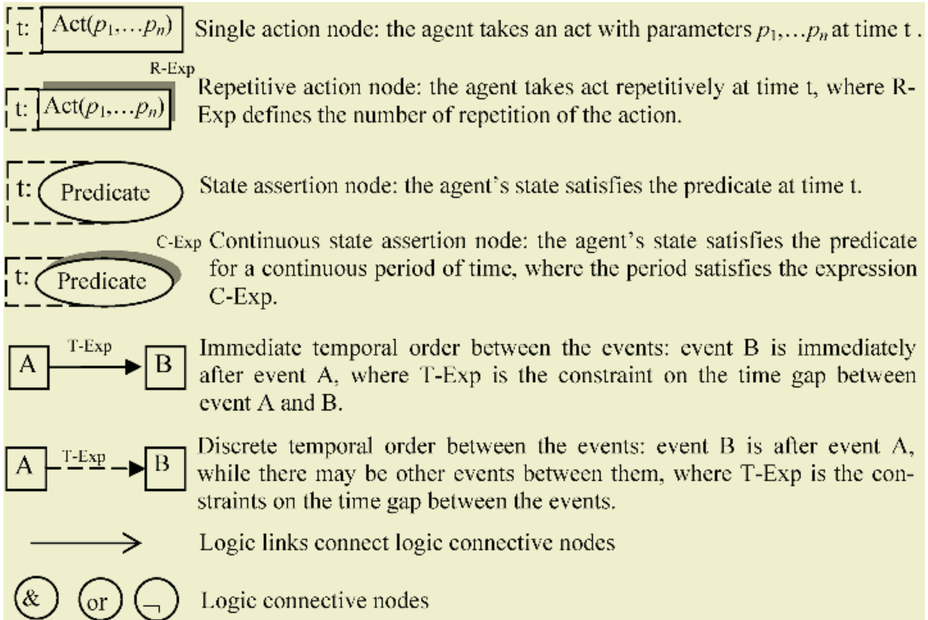


Fig. 10. Notations of Scenario Diagram

For example, Fig. 11 describes a scenario where Greenspan announces that the interest rate will increase by 0.25 points and all stock market analysts recommend sell Microsoft's share.

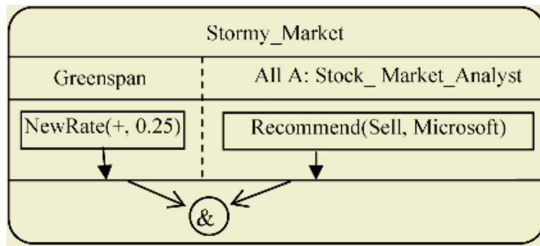


Fig. 11. Example of scenario diagram

3.3.2 Behaviour Diagrams. Behaviour diagrams describe agents' designed behaviour in certain scenarios. For each caste, a behaviour diagram defines a set of behaviour rules. Each rule describes how the agent of the caste should respond to a particular situation in the environment (i.e. in a scenario). The notation of behaviour diagrams includes the notation of scenario diagrams plus those in Fig. 12.

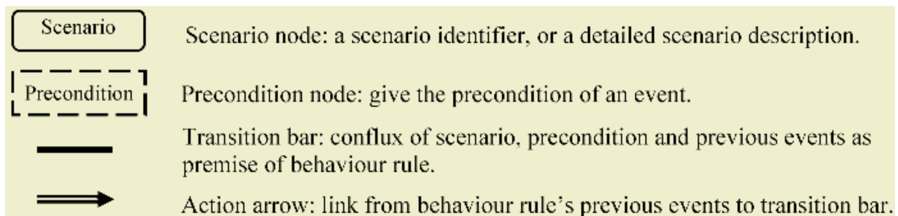


Fig. 12. Notation for behaviour diagrams

A behaviour diagram contains event nodes linked together by the temporal ordering arrows as in scenario diagrams to specify the agent's previous behaviour pattern. A transition bar with a conflux of scenario, precondition and previous pattern and followed by an event node indicates that when the agent's behaviour matches the previous pattern and the system is in the scenario and the precondition is true, the event specified by the event node under the transition bar will be taken by the agent. In a behaviour diagram, a reference to a scenario indicated by a scenario node can be replaced by a scenario diagram if it improves the readability. The behaviour diagram in Fig. 13 partly defines the behaviour of an undergraduate student. It states that if the student is in the final year and the average grade is 'A', the student may request a reference from the personal tutor for the application of a graduate course. If the personal tutor agrees to be a referee, the student may apply for a graduate course. If the department office offers a position in a graduate course, the student will join the Graduates caste and retreat from the Undergraduates caste.

In CAMLE language, each agent/caste has a designated environment, which is defined by its environment description. Therefore, in the development of a behaviour model for a given agent/caste, the modeller needs not to know all agents in the system, but only those in the environment. The modeller also does not need to know the full details of the behaviour of the agents in its environment, but just their capabilities in terms of the actions that they can take. As shown in the above example, the modeller needs not to know how a faculty member makes decisions on whether or not to write a reference for his/her tutee, and how the department decides who will be ac-

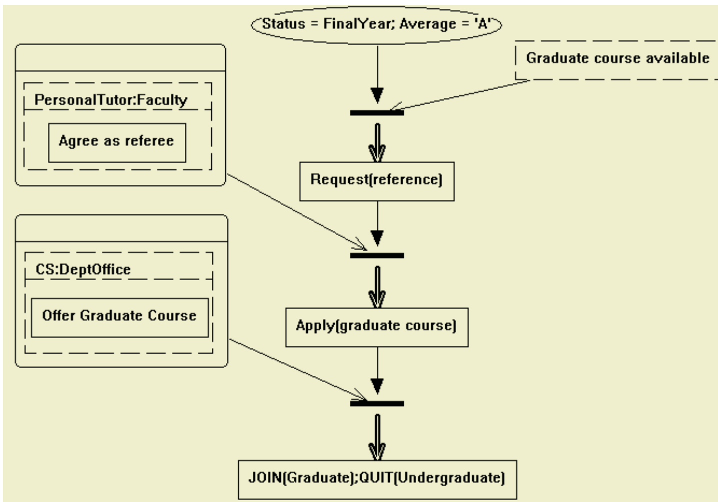


Fig. 13. Behaviour Diagram for Undergraduate Student

cepted as a postgraduate student. What the modeller only needs to know is that the personal tutor is capable of writing a reference letter and the department is capable of make a decision on the intake of postgraduate students[1]. We believe that this is perhaps the minimal amount of information that a modeller need to know about the system that his component will collaborate with.

3.4 Consistency of Models

Like all multiple view modelling languages, a model in CAMLE may have inconsistency between various types of models. Errors such as ill-formed diagrams may be introduced. Furthermore, because there are overlaps of information in different models, inconsistency between models can occur. In the case studies of CAMLE language and the modelling environment, we found the following types of errors and inconsistencies are among the most common problems.

(1) The same agent or caste may be refereed to with slightly different names.
(2) The set of actions occurred in the specific collaboration diagrams that specify various collaboration scenarios often does not match the set of actions declared in the corresponding general collaboration diagram.
(3) The names and parameter types of the actions defined for one caste/agent often do not match the references to these actions in scenario diagrams and behaviour diagrams.
(4) The behaviour of an agent or caste as described in a collaboration model is often inconsistent with what is defined in the behaviour model. For example, a collaboration model requires an agent to have certain sequence of activities, but the behaviour model does not define a corresponding behaviour rule or rules. It is also common that a behaviour rule requires the observation of an agent's visible action but the definition of the agent does not contain the action as a visible one.

---

[1] Of course, in a more complicated system, if the modeller does not have such knowledge, he/she needs to know how his/her agent can discover such knowledge at runtime.

In order to ensure the consistency between various models and models at different levels of abstraction, three types of the consistency constraints have been identified and formally defined in the CAMLE language. Automated tools are implemented in the modelling environment to check if these consistency constraints are satisfied. These consistency constraints including (A) well-formedness conditions imposed on each diagram, (B) intra-model consistency constraints that are imposed on diagrams of the same model at the same abstraction levels, and (C) inter-model consistency constraints that are imposed either on the same type of models at different abstraction levels, or on different types of models at the same level of abstraction. The definitions of the constraints are omitted here for the sake of space; see [15] for details.

4 Support Environment

A software environment to support the process of system analysis and modelling in CAMLE has been designed and implemented[2]. CAMLE aims at representing information systems naturally using the conceptual model of MAS presented in the previous section and facilitating the reasoning about such systems. It serves two interrelated purposes, i.e. to develop abstract descriptive models of current systems and to develop prescriptive designs of systems to be implemented. Therefore, in addition to model construction, two key features of the language and environment are regarded as of particular importance: (a) the consistency check between various models from different views and at different levels of abstraction, and (b) the transformation of diagrammatic models into formal specifications. Details of these functionalities are beyond the scope of this paper and are reported separately [15].

Fig 14 shows the architecture of the current CAMLE environment and its main functionality. The diagram editor supports the manual editing of models through graphic user interface. The well-formedness checker ensures that user entered models are well-formed, hence prevents syntactically incorrect diagrams from being processed. The partial diagram generator can generate partial models (incomplete diagrams) from existing diagrams to help users in model construction. It is based on the consistency constraints so that the generated partial diagrams are consistent with existing ones according to the consistency conditions. Consistency checking tools that help to ensure the well-formedness, consistency and completeness of system models are based on consistency constraints defined by the CAMLE language. Fig 15 is a screen snapshot of the output of the consistency checking tool. The diagnostic information helps users to locate and correct errors in the checked model.

The transformation from graphic models in CAMLE into formal specifications in SLABS enables engineers to analyse, verify and validate system models before the system is implemented. The specification generation tool in the CAMLE environment can automatically derive a formal specification in SLABS after a model is constructed and its consistency checked. Fig 16 shows a screen snapshot of the tool-generated specification of the caste Undergraduate.

Besides the University example used in this paper, a number of case studies of the modelling language and its modelling environment have been conducted. These case studies include the following.

---

[2] The tool is available for free for academic and research purposes. Please contact the authors.

Fig. 14. The Architecture of CAMLE Environment

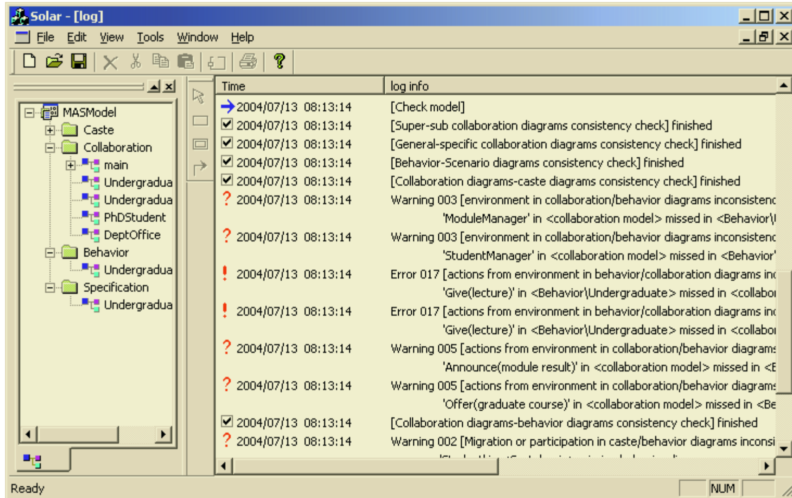


Fig. 15. Screen snapshot of the consistency checking tool's output

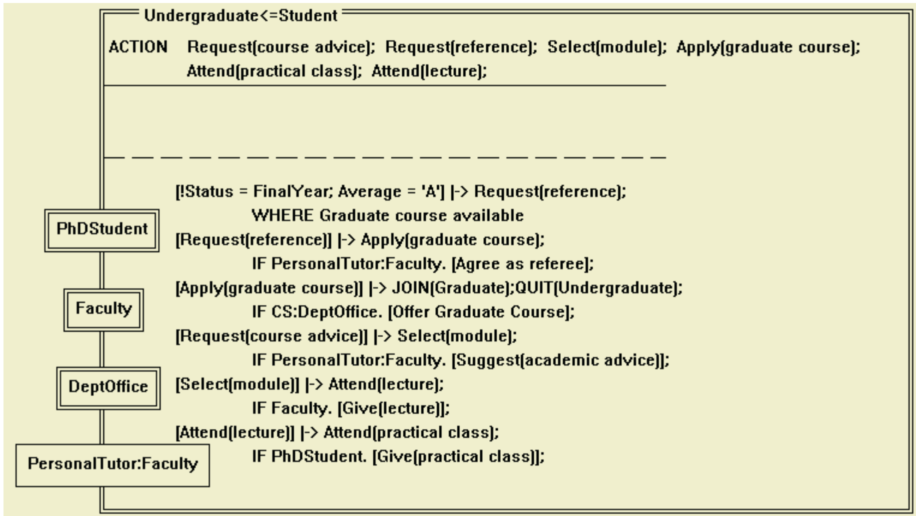


Fig. 16. Screen Snapshot of Generated Specification of the Caste Undergraduate

- *United Nations' Security Council:* The organisational structure and the work procedure to pass resolutions were modelled and a formal specification of the system in SLABS was generated. Details of the case study as well as modelling in other agent-oriented modelling notations can be found on AUML's website[3].
- *Amalthaea:* Amalthaea is an evolutionary multi-agent system developed at MIT's Media Lab to help the users to retrieve information from the Internet [16]. The system was modelled and a formal specification was generated.
- *Web Services:* The case study modelled the architecture of web services and an application of web services on online auctions. A formal specification in SLABS of the architecture and application was generated successfully. More details of the specification of web services in SLABS can be found in [17].

In the case studies, we found that the CAMLE language was highly expressive to model information systems' organisational structures, dynamic information processing procedures, individual decision making processes, and so on. Models in CAMLE were easy to understand because they naturally represent the real world systems. The automated environment greatly helped to manage the consistency between various diagrams from difference views and at different abstraction levels. It significantly reduced the difficulty in the development of formal specifications of multi-agent systems especially for complicated systems such as Amalthaea.

5 Conclusion

This paper presented the agent-oriented modelling language and environment CAMLE. It is based on the conceptual model of MAS of the formal specification language SLABS. Models represented in the CAMLE can be automatically checked

[3] URL: http://www.auml.org/

for consistency and transformed into formal specifications in SLABS by the tools in the modelling environment. The modelling language has a number of novel features, which include the congregate whole-part relation and migration relation between castes, the designated environment descriptions, scenario diagrams, scenario driven behaviour rules, and most importantly, the concept of caste.

There have been a number of efforts in the direction of AO methodology, many of which focus on the process of MAS engineering as well as the representation of MAS. With regard to conceptual model of the methodologies, there is a fundamental difference between CAMLE and the methodologies that are based on mental state related notions such as belief, desire, intention, goal and plan. Although these notions are widely used, their meanings vary from people to people in different methodologies. CAMLE replaces these notions with an abstract model of agents as encapsulation of data, operation and behaviour. CAMLE also has a fundamental distinction from the methodologies that are based on social organization related notions such as roles, agent society and organization structure. CAMLE replaces such intuitive concepts with a well-defined language facility caste, which is easy to understand and use from software engineering perspective. Caste can be used to represent a number of the concepts in agent-oriented modelling, such as roles, agent societies, normative behaviour, common knowledge and protocols, etc. [6]. The caste-centric feature enables us to achieve simplicity in the design of an expressive modelling language and efficiency in the implementation of the powerful environment.

Among the related work, Gaia is perhaps one of the most mature agent-oriented software development methodologies at the moment. It does not commit to specific notations for modelling concepts such as roles, environment and interaction [1]. UML notation are widely used, e.g. in Tropos, PASSI [18] and AUML [19]. However, there is no clearly defined conceptual model or meta-model underlying the uses of UML notations for agent-oriented modelling although there are fundamental differences between agents and objects as discussed in section 2 and 3.

A related work on agent-oriented modelling language is ANote [20], which also provides a set of diagrams to model different views of MAS. Systems' structural aspects, dynamic aspects and physical aspects are specified with the notations representing the concepts of goal, agent, ontology, scenario, planning, interaction and organization etc. Although the concepts and notations are different from the ones used in CAMLE, we share the same opinion that using (modified) OO paradigm to model agent-based system is not desirable.

There are several issues remaining for future work. We are investigating software tools that support model-based implementation of MAS in CAMLE. The design and implementation of an agent-oriented programming language with caste as the basic program unit is on the top of our agenda.

Acknowledgement

The work reported in this paper is supported by China High-Technology R&D Programme under the grant 2002AA116070.

References

1. Dam, K. H., Winikoff, M., "Comparing Agent-Oriented Methodologies", *Proc. of AOIS'03,* Melbourne, Australia, July 2003.
2. Zambonelli, F., Jennings, NR and Wooldridge, M., "Developing multiagent systems: the Gaia Methodology", *ACM TOSEM* 12(3), 2003, pp. 317-370.
3. Bresciani, P., Giorgini, P., Giunchiglia, F., Mylopoulos J. and Perini, A., "TROPOS: An Agent-Oriented Software Development Methodology", *Journal of Autonomous Agents and Multi-Agent Systems* 8(3), Kluwer Academic Publishers, May 2004, pp. 203 - 236.
4. Zhu, H., "Formal Specification of Agent Behaviour through Environment Scenarios", *Formal Aspects of Agent-Based Systems*, Rash, J.L., *et al.*, (Eds.), Springer, LNCS Vol. 1871, 2001, pp. 263-277.
5. Zhu, H., "SLABS: A Formal Specification Language for Agent-Based Systems", *Int. J. of Software Engineering and Knowledge Engineering* 11(5), 2001, pp. 529-558.
6. Zhu, H., "The role of caste in formal specification of MAS", *Intelligent Agents: Specification, Modelling, and Application, Proc. of PRIMA'01*, Yuan, S-T; Yokoo, M. (Eds.), LNCS, Vol. 2132, Springer, 2001, pp.1-15.
7. Zhu, H., "Formal Specification of Evolutionary Software Agents", *Formal Methods and Software Engineering*, *Proc. of ICFEM'2002*, George, C. and Miao, H., (Eds.), LNCS, Vol. 2495, Springer, 2002, pp.249~261.
8. Shan, L. and Zhu, H., "Analysing and Specifying Scenarios and Agent Behaviours", *Proc. of IAT'03,* Halifax, Canada, Oct. 2003.
9. Shan, L. and Zhu, H., "Modelling Cooperative Multi-Agent Systems", *Proc. of the 2nd Int. Workshop on Grid and Cooperative Computing*. Shanghai, China. Dec. 2003.
10. Drossopoulou, S., Damiani, F., Dezani-Ciancaglini, M. and Giannini, P., "More dynamic object reclassification: $\text{Fickle}_{\text{II}}$", *ACM TOPLAS,* 24(2), 2002, pp. 153-191.
11. Zhu, H., and Lightfoot, D., "Caste: A step beyond object orientation", *Modular Programming Languages, Proc. of JMLC'2003*, Boszormenyi, L., & Schojer, P. (eds), LNCS Vol. 2789, Springer, 2003, pp.59-62.
12. Shen, R., Wang, J. and Zhu, H., "Scenario Mechanism in Agent-Oriented Programming", *Proc. of APSEC'04*, Oct 30-Dec 3, 2004, Busan, Korea, *in press*.
13. Odell, J., Parunak, H. V. D. and Fleischer, M., "The Role of Roles", *Journal of Object Technology* 2(1), 2002, pp.39-51.
14. Barbier, F., Henderson-Sellers, B., Le Parc A. and Bruel J-M., "Formalization of the Whole-Part Relationship in the Unified Modelling Language", *IEEE TSE* 29(5), 2003, pp.459-470.
15. Shan, L. and Zhu, H., "Consistency Check in Modeling Multi-Agent Systems", *Proc. of COMPSAC'04*, Hong Kong, IEEE CS, Sept., 2004.
16. Moukas, A., "Amalthaea: Information Discovery and Filtering Using a Multi-Agent Evolving Ecosystem", *Journal of Applied Artificial Intelligence*, 11(5), 1997, pp.437 457.
17. Zhu, H., Bin Zhou, B., Mao, X., Shan, L., and Duce, D., "Agent-Oriented Formal Specification of Web Services", *Proc. of the AAC-GEVO'04 at GCC'04*, LNCS Vol. 3252, Springer, Oct. 2004.
18. Burrafato, P. and Cossentino, M., **"**Designing a Multi-Agent Solution for a Bookstore With the PASSI Methodology", *Proc.* of AOIS'02 at CAiSE'02, May 2002.
19. Bauer, B., Muller, J.P. and Odell, J., "Agent UML: A Formalism for Specifying Multiagent Software Systems", *Agent-Oriented Software Engineering*, Ciancarini, P. and Wooldridge, M. (eds.), LNCS, Vol. 1957, Springer, 2001, pp.91-103.
20. Shan, L. and Zhu, H., "CAMLE: A Caste-Centric Agent Modelling Language and Environment", *Proc. of SELMAS'*04 *at ICSE 2004*, Edinburgh, Scotland, UK, May 2004.

A Formal Approach for the Modelling and Verification of Multiagent Plans Based on Model Checking and Petri Nets

Hyggo Oliveira de Almeida[1], Leandro Dias da Silva[1], Angelo Perkusich[1], and Evandro de Barros Costa[2]

[1] Electrical Engineering Department, Federal University of Campina Grande, Postal Code 10.105 – 58109-970, Campina Grande, Paraíba, Brazil
{hyggo,leandro,perkusic}@dee.ufcg.edu.br
[2] Information Technology Department, Federal University of Alagoas, Maceió, Alagoas, Brazil
evandro@tci.ufal.br

Abstract. Multiagent systems are characterized by decentralized control and agents that perform autonomous actions. The sequence of such actions are generally described by plans. An important issue in this context is how to verify the correctness of plans when agents have unpredicted actions. In this paper, formal modelling and verification guidelines to verify nondeterministic multiagent system plans are introduced. The guidelines are based on HCPN modelling, simulation, and model checking. The guidelines are conceptually introduced, and then applied for a multiagent intelligent tutoring system modelling and verification.

1 Introduction

Multiagent Systems (MAS) have become a promising approach to develop complex software systems [1]. Usually, the sequence of actions that agents have to execute is defined by a plan. Planning is the process of generating a plan based on three essential inputs: an initial state of the world; a set of possible actions that an agent can execute; and a set of goals to be reached. Application areas of multiagent planning include multi-robot environments [2], cooperating Internet agents [3], logistics [4], manufacturing systems [5], and military tasks [6], among others.

In the context of planning, one key issue is how to verify the correctness of multiagent plans. Due to the inherent decentralized nature of MAS, agents have only partial knowledge of the environment, and thus it is difficult to define deterministic choices based on global information [7]. Usually, agent plans are built without considering uncontrollable and conflicting actions of other agents. Based on such actions and on a nondeterministic plan, one can define a set of next states instead of only one for the deterministic case [8]. Therefore, adding flexibility for the plan execution.

However, the generation or verification of nondeterministic plans may lead to the well known state explosion problem. Thus, there is no single planning algorithm appropriate for all systems of agents. Many plan verification methods are described in the multiagent literature but they are efficient for some agent systems and inefficient for

R. Choren et al. (Eds.): SELMAS 2004, LNCS 3390, pp. 162–179, 2005.
© Springer-Verlag Berlin Heidelberg 2005

others. A possible approach to deal with such situation is to adopt efficient techniques for either plan generation or verification [9].

In this work model checking [10] is used to deal with the complexity associated with the validation of nondeterministic plans for multiagent systems [11, 12]. Also, techniques, such as simulation and message sequence charts (MSC) [13] have been applied. In order to efficiently and systematically integrate these techniques, guidelines to model and verify plans are introduced. Hierarchical Colored Petri Nets (HCPN) [14, 15] are applied to model multiagent systems. In order to illustrate the use of these guidelines, the verification of plans for a multiagent intelligent tutoring system is presented.

The remaining of this paper is organized as follows. In Sections 2 and 3, the modelling and verification guidelines are introduced. Section 4 presents the case study based on a multiagent intelligent tutoring system. The application of the guidelines for the case study is presented in Section 5. In Section 6 related works are discussed. Finally, in Section 7 final remarks are presented.

2 Modelling Guidelines

In this section, the guidelines to obtain a Hierarchical Colored Petri Net model for multiagent systems are presented. As discussed in the introduction, these guidelines allow a designer to obtain the model needed to apply the verification guidelines that are introduced in Section 3. In Figure 1 the guidelines are illustrated, and are described as follows.

Modelling Guideline 1 – Identify Types of Architectures for Agents

Depending on the application domain, a multiagent system may be composed by agents with different internal architectures [16]. On the other hand, agents implementing the

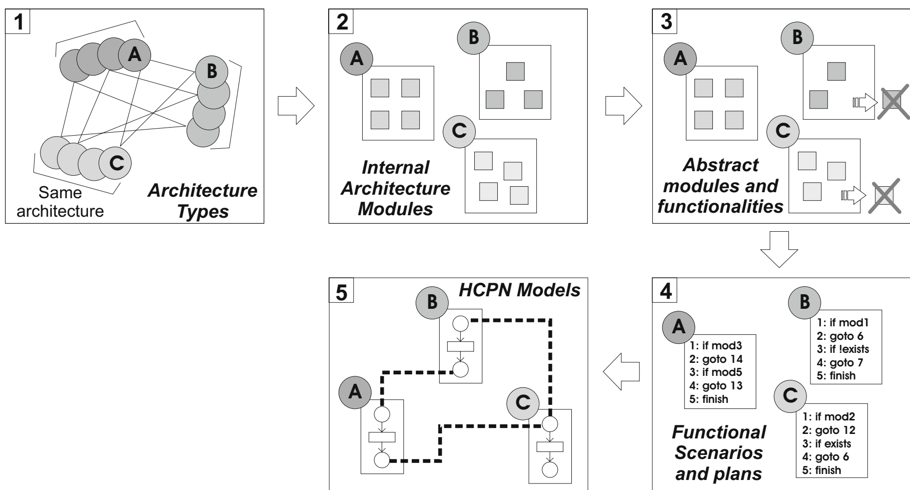


Fig. 1. Guidelines for modelling multiagent plans.

same functionalities usually have the same architecture. For instance, consider an information retrieval multiagent system. Generally speaking, one such system is composed by agents with different functionalities, namely: agents for information extraction, filtering, searching, and so on. Thus, the designer must identify different types of architectures. The result of this guideline is a set of types of architectures for the agents. As explained latter on, this guideline is necessary in order to verify the plans for different types of architectures.

Modelling Guideline 2 – Group Functionalities for Agents in Modules for Each Architecture Type

One major characteristic of the multiagent approach is the independence of the internal agent development paradigm. In the same way as many software engineering paradigms, a common approach is to group functionalities into modules. Therefore, the verification of properties defined for such modules can be performed locally.

If the internal agent architectures previously identified are not constructed based on a modular based infrastructure, the functionalities of agents must be grouped in modules for each type of architecture, as defined for the Guideline 1. This is important to build plans describing high level modules as providers of functionalities. Thus, it is possible to point out which module or functionality does not satisfy a specific property.

Modelling Guideline 3 – Define Functionalities and Modules for Modelling

In a multiagent system, some features are generally shared by all internal agent architectures. Depending on the properties to be verified, some functionalities can be abstracted from the model. Examples of such features are related to agent communication, which can be omitted if there is no security, performance, or protocol-related issues to verify. In order to make the model more concise and the verification more efficient, the designer must define which functionalities and modules must be modelled in order to proceed to the verification guideline.

Modelling Guideline 4 – Define/Identify Agent Plans

As mentioned before, the sequence of agent actions in multiagent systems are usually described by plans. If the system being modelled is executed based on agent plans, then skip this guideline. If there are no plans defined for each agent, build execution scenarios. This is necessary to define agent execution algorithms in the first verification guideline, as detailed in Section 3. In some cases, agents with the same architecture have also the same plan. Scenarios should be described considering modules and functionalities identified in the previous guideline.

Modelling Guideline 5 – Construct Hierarchical Colored Petri Net Models for Each Agent Architecture

Construct a Hierarchical Colored Petri Net (HCPN) model for each type of architecture previously defined, as discussed for Guideline 1. The most important characteristic is that the model must capture the internal behavior of the agents, according to their architecture.

After the application of modelling guidelines, an HCPN model for each architecture and a description for each agent plan is obtained. The models together with the plans are then used for the verification, according to the guidelines that are explained as follows.

3 Verification Guidelines

A formal model of a system allows its verification before it is built in order to determine design problems, or can be used to improve an existing one. There are several ways to analyze a system, such as simulation and model checking, among others. For HCPN models, the tool set Design/CPN [17] is adopted, in the context of this work, for graphical edition, simulation, automatic generation of Message Sequence Charts (MSC), and to perform model checking. The simulation gives the designer the insight about the behavior of the system. The generated MSCs are useful to observe different execution traces for the models, and abstracting the token game of a Petri net simulation [14]. An MSC automatically generated during a simulation run is then used to define predicates related to the model that are necessary to perform model checking. Thus, these predicates are used to prove desired properties for a given plan.

It is important to observe that simulation does not guarantee that the system always behaves as expected, since only one possible execution trace is captured. Therefore it is necessary to verify the plan for all possible behaviors, or traces. To do so, model checking is applied. The model checking technique verifies if a model $\mathcal{M}$ satisfies a given specification f, that is denoted as: $\mathcal{M} \models f$ [10]. In the context of this work the model is an HCPN, and the specification is a temporal logic formula. The library ASK/CTL [18] for the Design/CPN is used to perform model checking. The temporal logic formulae specify the desired scenarios. The model checking is performed on the state space for the HCPN model, generated using the Design/CPN toolkit, to verify whether the desired properties are satisfied by the HCPN model or not. In the following it is discussed how the plans for each HCPN model for an agent is analyzed.

In order to prove the correctness of the plan a set of verification guidelines is defined. These guidelines favor a systematic proof method because the designer can use the same reasoning for every verification procedure in the context of plan verification for multiagent systems. In Figure 2 the guidelines are illustrated, and are detailed as follows.

Verification Guideline 1 – Describe the Overall Execution Plan

Define an abstract description of the behavior of each agent. The idea is to describe the high level behavior of the agent without considering specific activities. For this guideline, *what* the agent does is defined. An algorithm based notation can be used to describe the execution plan, as adopted in this work. In some situations this description can be too complex to be verified at once. Nevertheless, they are useful for the Guideline 2, as detailed bellow.

Verification Guideline 2 – Identify Specific Plans

The overall execution plan description can then be divided into specific plans for specific activities. This is a suitable way to verify multiagent plans because it is possible

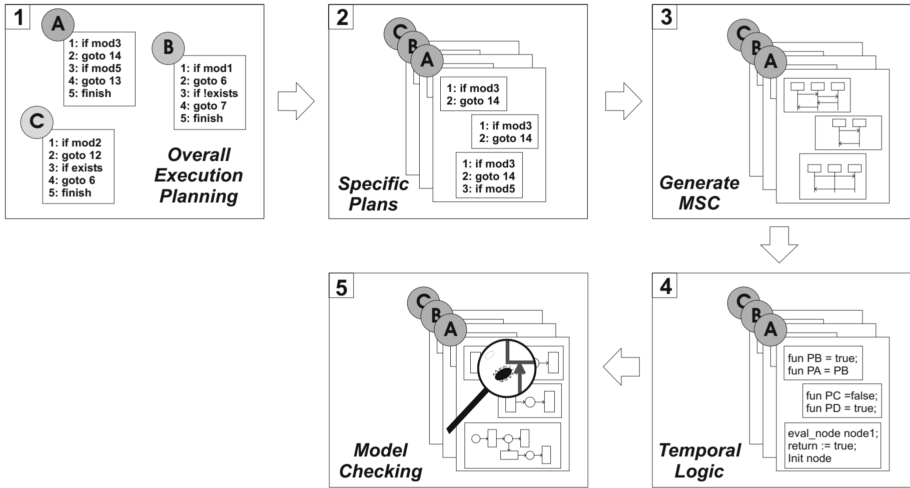


Fig. 2. Guidelines for the verification of multiagent plans.

to reduce the size of the formulae to be verified. Also, it is easier to find and fix errors since the specific activities are performed for each module. Thus, making possible to identify exactly which module is not behaving properly. In the case study presented in Section 4, the execution plan of an agent can be divided in the specific component plans. Again, an algorithm based notation is used to describe specific plans.

Verification Guideline 3 – Generate the MSC for the Plans

As said in the beginning of this section the MSC is useful to illustrate the execution trace in a more intuitive way. Also, usually the Petri net model is a complex model. Therefore it is necessary to generate the MSC for each specific plan identified on the previous guideline. These MSCs define the sequence of actions for a given plan and which modules perform the actions.

Verification Guideline 4 – Specify the MSC in Temporal Logic

Based on the MSC the designer can specify the behavior using temporal logic, in order to prove whether this behavior holds for every possible behavior or not. That is because the MSC is generated at simulation time, and it covers only one possible path. The specification is constructed using atomic propositions and temporal modalities and quantifiers to form temporal logic formulae.

Verification Guideline 5 – Perform Model Checking

The last guideline is to perform model checking to verify whether the model models the specified properties or not. As said before, the model checking is performed using the Design/CPN and the library ASK/CTL. First, the state space, that is called occurrence

graph, is generated for the model. The occurrence graph is a directed graph that represents all the possible behaviors of a Petri net model. After that, the model checking is executed to verify if the specification in temporal logic defined in the previous guideline is satisfied in the state space. If the specification is satisfied, the plan is proved correct. Otherwise there can be an error but as the plan is specific for an activity, and the activities are related to modules, it is straightforward to locate the error source. Moreover it is possible to use the counterexample to identify exactly for which sequence of actions the specification is violated [10].

The guidelines 2, 3, 4, and 5 can be performed one-by-one for each plan, or for all plans before proceeding to the next guideline. When analyzing systems for complex domains where there are several entities communicating with each other or executing independently, it is necessary to use several tools. For the verification guidelines defined in this section, simulation, message sequence charts, and model checking are used in an organized and systematic way to verify execution plans for agents' activities. Each tool is used to solve an specific analysis problem. To know, the MSC is used to identify exactly what flow, in exactly which place must be analyzed. This is done by simulation. Without the MSC must be difficult to identify the properties because the model is usually complex. But the simulation follows only one possible path. Therefore it is performed model checking to prove the properties for all possible behaviors. Using these tools in an integrated way makes the analysis activity easier.

4 Multiagent Intelligent Tutoring System

In this section the architecture of a multiagent intelligent tutoring system is presented. Such system is based on problem solving cooperative based learning activities, such as problem resolution, instruction, hints, and explanations. The architecture is based on the MATHEMA architecture [19] that has been applied in various domains, such as algebra [19] and musical harmony [20]. In Figure 3 the high-level architecture of the system is shown, and main actors are:

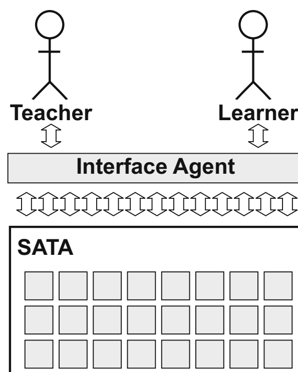


Fig. 3. High level architecture for the intelligent tutoring system.

- **Learner:** a human agent involved in a learning activity for a given knowledge domain.
- **Teacher:** a human agent to aid and facilitate the interaction between the learner and the system.
- **SATA (Society of Artificial Tutoring Agents):** implements the mechanisms to promote the successful interaction between the learner and the teacher. The SATA defines the multiagent intelligent tutoring system.
- **Interface agent:** promotes the interaction between the learner, the teacher, and the SATA.

In this section the focus is the architecture for an agent in the SATA. Each agent in such architecture is composed by three systems: tutoring, social, and distribution. The social and distribution systems implement the functionalities to promote the interaction among the agents. These systems are not discussed in this paper. The reader may refer to [19] for a detailed presentation.

The tutoring system (TS) implements the mechanisms to promote the cooperative interactions between an agent and the learner during the learning process. It is the "intelligent system" of an agent and is composed by the following entities.

Mediator

Mediator implements the interaction mechanisms with the Interface Agent and thus with the learner. It also selects the suitable reasoner in order to solve the task defined by the learner.

Reasoners

Reasoners are the components that implement the pedagogic functionalities of the intelligent tutoring system. The reasoners are organized as container modules that compose the tutoring system of an agent defined for the SATA, such as: *tutor module*, *expert module* and the *leaner modelling module*. These three modules are explained in the following.

- **Tutor module:** reasoners belonging to this module implement the pedagogical interactions with the learner.
  - *Pedagogical tasks manager* selects the resources that are available to the learner.
  - *Problem solver* replies to the questions of the learner.
  - *Evaluator* evaluates the answers given by the users.
  - *Remediator* defines the next system action in order to improve the performance of the learner based on a cognitive diagnosis.
- **Expert module:** reasoners that belong to this module implement the subjective problem evaluation for the learner. Its response is inferred based on production rules stored in a knowledge base. This functionality is provided by the reasoner *inference engine*.
- **Leaner modelling module:** reasoners that belong to this module are responsible to acquire, maintain, and represent the individual information about the learners. Such information are then used to define the best teaching strategy to be applied for a given learner.

- *History Manager* is responsible for the organization and storing of all pedagogical knowledge already demonstrated to the learner.
- *Profile Manager* is responsible for the management of the learning level for each learner for a given topic.
- *Cognitive Diagnostic* is responsible for the definition of the cognitive level of each learner according to the profile.

Resources Base

Resources base is the module that makes available the pedagogical resources, production rules and models for learners. It is composed by the following components. The *Resource Access Manager* is responsible to provide access to the resource base and its repositories. Such repositories are described below.

– *pedagogical knowledge:* resources such as definitions, examples, exercises, and hints.
– *learner model:* information related to the profile of the learner and the resources already studied.
– *knowledge bases:* inference rules used by the inference engine to evaluate the answers given by the learner.

5 Applying the Guidelines to the Case Study

5.1 Modelling Guidelines

The application of the modelling guidelines introduced in Section 2 for the case study described in Section 4 is presented below.

Modelling Guideline 1 – Identifying Types of Agent Architectures That Compose the Multiagent System. In the ITS case study, all agents described in Section 4 have the same internal architecture. Thus, there is only one type of architecture. The result of the application of this guideline is the architecture shown in Figure 4.

Modelling Guideline 2 – Grouping Agent Functionalities in Modules for Each Architecture Type Defined in the Previous Guideline. As said before, the architecture of the agents for the ITS are composed by three systems (tutoring, social and distribution), which have specific functionalities. Each system are defined as a composition of more specific modules (See Figure 4). These modules are the result of this guideline.

Modelling Guideline 3 – Abstracting the Modules That Implement Functionalities Which Are Not Relevant to the Verification. For the ITS case study, modules that implement communication and interaction functionalities are abstracted from the model. Therefore, only modules of the tutoring system are considered (see Figure 4). The reader may refer to [21] for a detailed discussion concerning to communication and interaction modelling and verification of an ITS multiagent system.

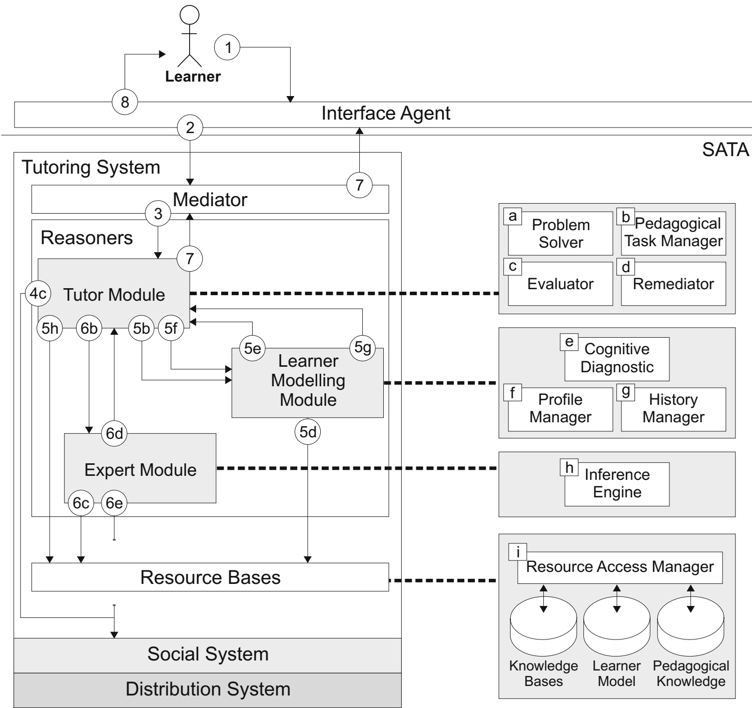


Fig. 4. Functional view for the tutoring system.

Modelling Guideline 4 – Describing a Plan or a Functional Scenario for Agents, Considering the Modules Previously Defined. In Figure 4, it is illustrated the functional view of an agent taking into account the interactions with other agents and with the learner. The functional scenario starts when a learner asks to the multiagent intelligent tutoring system for a resource. It can be a definition, an example, an exercise, a hint, the solution of a proposed problem or the evaluation of an answer for a problem proposed to learner by the system. Based on this initial state for the scenario the sequence of events and interactions are discussed in the following.

1. The learner asks the interface agent for a resource of a pedagogical unit.
2. The interface agent selects an agent in the SATA responsible for pedagogical unit and send the requisition to it. The selected agent is the supervisor agent for the current actions.
3. The mediator of the agent receives the requisition and selects the reasoner to execute the requested task.
4. When the learner asks for a problem resolution proposed by him, the reasoner selects a problem solver (see the problem solver box labelled with A in Figure 4) that executes the following steps:
   (a) It divides the problem to be solved into smaller sub-problems.
   (b) Solves each sub-problem.
   (c) When there are sub-problems it cannot solve, it forwards them to the social system. The social system allocates other agents to solve them.

   (d) When all the sub-problems are solved the answer is then returned to the mediator (see step 7).
5. If the learner asked for a resource, the selected reasoner is then the pedagogical tasks manager (label B in Figure 4), and then executes the following steps:
   (a) Asks the mediator (D) for the next type of resource to be presented to the learner according to his cognitive profile.
   (b) The mediator asks the profile manager (F) to recover the level of the learner with respect to the pedagogical unit being studied.
   (c) The profile manager asks the history manager (G) for the history of resources already given to the learner for the current pedagogical unit.
   (d) The history manager asks the resources access manager (I) for the list of resources already given by the learner and return it to the history manager, that returns it to the profile manager.
   (e) Based on the learner history, the profile manager defines his quantitative knowledge level, based on the previous scores obtained for the other resources belonging to the current pedagogical unit. Then, it returns this score to the remediator.
   (f) The remediator gets the quantitative score and forwards it to the cognitive diagnostic (E) so that it can identify the qualitative level of the learner with respect to the current pedagogical unit, such as, for example, basic, intermediary, or advanced.
   (g) The cognitive diagnostic returns the qualitative level of the learner to the remediator, that defines the type of resource to be given to the learner and forwards them, the level "n" and resource "t", to the pedagogical tasks manager.
   (h) The pedagogical tasks manager asks for a resource "t" and level "n" to the resources access manager and returns the resource to the mediator (see step 7).
6. If the leaner asked for an evaluation of his answer to a problem previously given by the system (what would occur in step 5) the selected reasoner is evaluator (C), that thus executes the following steps:
   (a) If the problem is an objective one, the evaluator verifies if the correct answer to the question is the same as the one given by the learner and returns the score to the mediator (step 7).
   (b) In the case that the problem is subjective, it is forwarded to the inference engine (H) that divides it in smaller parts.
   (c) The inference engine then asks to the resources access manager for knowledge inference rules and tries to validate each part of the solution.
   (d) When the parts are validated, the whole solution is then validated. Therefore, the final score is returned to the evaluator.
   (e) If some parts cannot be validated, they are forwarded to the social system, and then other agents are allocated to evaluate them.
   (f) The evaluator then updates the history of the learner and returns the score to the mediator.
7. Based on the answer given to the learner, the mediator forwards it to the interface agent.
8. The interface agent then forwards it to the learner: the resolution of the problem proposed by the learner, a resource or the evaluation of the answer for a problem proposed by the system together with the score.

Modelling Guideline 5 – Constructing an HCPN Model for Each Agent Architecture. The application of this guideline is presented in [21], where the complete Hierarchical Colored Petri Net model for the ITS case study is presented. In Figure 5, the hierarchy page for the model is shown. It has been graphically organized to be similar to the architecture diagram shown in Figure 4 and explained in Section 4, and therefore it is not detailed.

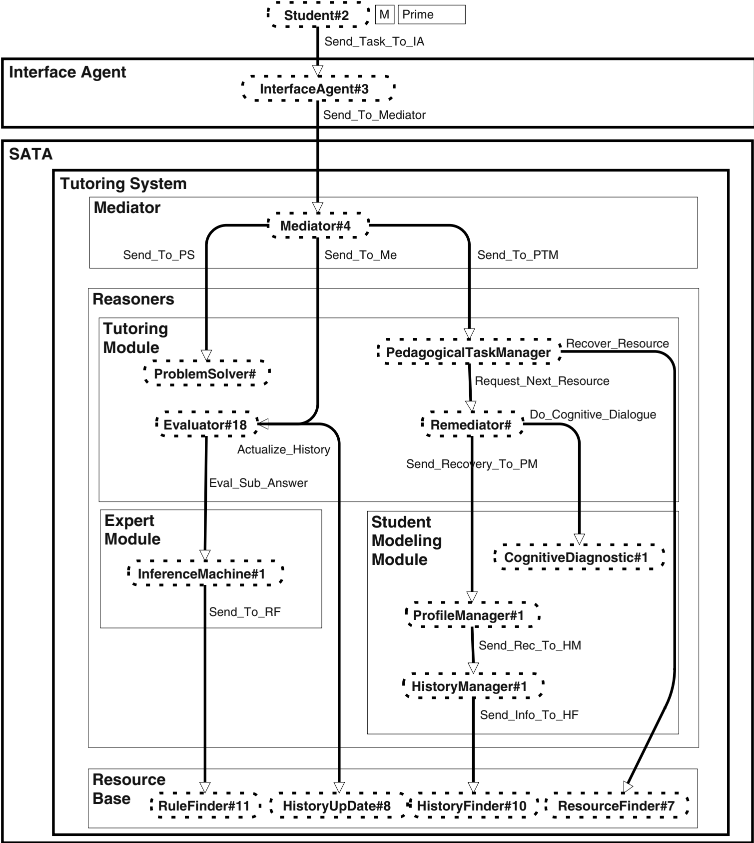


Fig. 5. Hierarchy page for the ITS agent Colored Petri Net Model.

5.2 Verification Guidelines

In the following the guidelines to validate the agent plan represented by model shown in Figure 5 are applied.

Verification Guideline 1 – Defining the Overall Plan for the Agents. Based on the functional scenario previously described, an algorithm-like style is used to present the overall plan for the ITS agents (Algorithm 1).

Algorithm 1 Overall Execution Plan.

1: Verify the learning level of the student
2: **if** The desired level has reached **then**
3: Go to step 17
4: **else**
5: Define the new step p
6: **end if**
7: **if** p is an explanation or an example **then**
8: Show it
9: Go to step 15
10: **else**
11: Show an exercise
12: **end if**
13: Receive the answer from student and try to validate it
14: Define an score to the exercise according to the validation
15: Increment the learning level
16: Go to step 1
17: Inform the student that this topic is over and pass the control to another agent

Verification Guideline 2 – Defining Specific Plans for the Agents. Based on the overall plan previously defined, the specific plans for the ITS agents are described below.

Specific Plan 1 – Suppose that the student requests the system to solve some problem. This is the simplest type of interaction between them. In this case, the system do not try to verify the learning level of the student, neither to increment it after the end of the activity. By this request the student is using the system as a problem solver. The module called *Problem Solver* processes this request as follows. Every agent has the knowledge in a specific expertise, and when the problem can be solved by the current agent, the request is processed. On the other hand, the agent requests a cooperation with another agent that can solve the problem. In either case, the answer is returned to the student.

Consider the first order equation $3x/4 + 5x/3 = 10$. In order to solve it, an agent must ask for a cooperation with another agent that knows how to calculate the least common multiplier between them. This is a simple example that is useful to see how the system works. Therefore, according to Algorithm 1, the part of the plan to deal with a problem proposed by the student is show in Algorithm 2. In this plan, the *Interface Agent* and the *Mediator* module are omitted because they are used in all interactions. Therefore they are implicitly considered, for the sake of simplicity and space limitations.

Specific Plan 2 – Suppose the situation where the student must solve a problem and send it to system to analyze his answer. In this scenario, the reasoner used is the *Evaluator*. In Algorithm 3 the plan for this interaction is specified.

Specific Plan 3 – The last possibility is when the student needs a resource. A resource can be, for instance, an explanation, or a definition, for example. This is the most complicated plan because several modules are used to execute it. In Algorithm 4, the definition of this plan is shown. For this algorithm it is used a name convention for modules,

Algorithm 2 Problem Solver Plan.

1: **if** Problem can be solved locally **then**
2: Solve it
3: **else**
4: Ask for cooperation
5: **end if**
6: Return the result to the student

Algorithm 3 Evaluator Plan.

1: **if** Problem is objective **then**
2: Compare the answer with stored result
3: Go to step 13
4: **end if**
5: Call the *Inference Engine* module
6: Divide the subjective problem into smaller ones
7: Try to validate each part using inference rules in the resource base
8: **if** All parts can be validated **then**
9: Go to step 13
10: **else**
11: Ask for cooperation
12: **end if**
13: Give a score
14: Update the history
15: Return the score to the student

where PTM represents *Pedagogical Task Manager*, Rem represents *Remediator*, PM represents *Profile Manager*, HM represents *History Manager*, and CD represents *Cognitive Diagnostic*.

Verification Guideline 3 – Generating an MSC for Each Specific Plan. In Figure 6 it is shown the MSC for the problem solver specific plan, that is described in Algorithm 2.

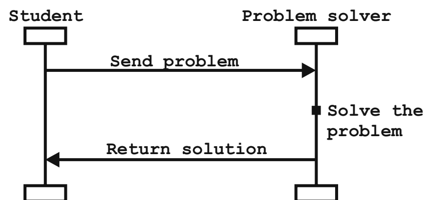


Fig. 6. Execution plan for the problem solver.

Algorithm 3 is quite more complicated then the previous one. Therefore the importance of the use of MSC is clear for this case. The message sequence charts that represent the plans to analyze subjective and objective problems are shown in Figure 7 and 8, respectively. They are related to the evaluation of the specific plan described in Algorithm 3.

Algorithm 4 Pedagogical Task Manager Plan.

1: PTM asks the student level to Rem
2: Rem asks the quantitative level to PM
3: PM asks the student history to HM
4: HM gets the history from the resource base
5: HM sends history back to PM
6: PM define the student profile
7: PM sends profile back to Rem
8: Rem asks the qualitative level to CD
9: CD define the next resource and the level
10: CD sends resource and level back to Rem
11: Rem asks the resource to resource base
12: Rem sends the resource back to PTM
13: PTM sends the resource back to student.

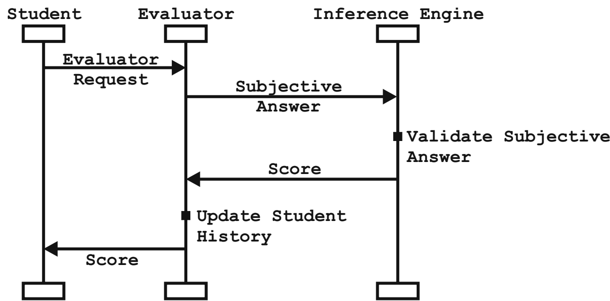


Fig. 7. Execution plan for the subjective problem evaluation.

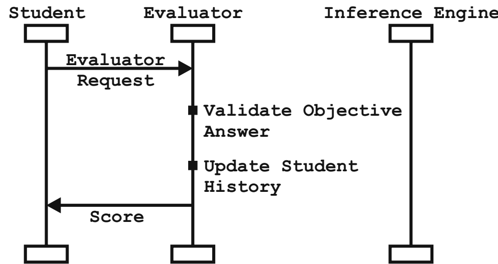


Fig. 8. Execution plan for the objective problem evaluation.

In the case of an objective problem solved by the student, the system must compare the given answer with the correct answer in order to give the student a score and to update his history and profile. The same holds for subjective problem. The difference in this case is that the inference machine must be used in order to apply specific rules to try to validate the answer.

Finally, the MSC of the plan related to the pedagogical task manager described in the Algorithm 4 is illustrated in Figure 9.

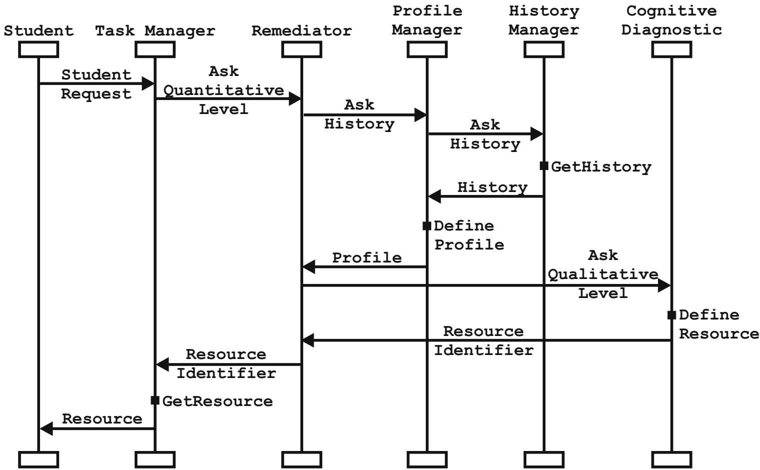


Fig. 9. Execution plan for the Pedagogical Task Manager.

Verification Guideline 4 – Specifying Temporal Logic Formulae. It is used here a simplified syntax for the propositions and formulae to improve readability. The actual formulae are more complicated due to the syntactic constructions of ASK/CTL and could be complicated for a reader not familiar with ASK/CTL to understand it.

The code for the atomic propositions specifying the properties defined for the MSC shown in Figure 6, is as follows. In this fragment of code *PA* is the atomic proposition that abstracts the fact that $STUDENT_1$ is sending a problem to the system. In the same way *PB* represents the fact that the Problem Solver module has been called. Finally *PC* represents that the answer has been sent back to student and the model reached the final state.

```
fun PA n=(Mark.Student'List_Task n=(STUDENT_1,[SEND_PROBLEM]));
fun PB n=(Mark.ProblemSolver'Solving n = (STUDENT_1,SEND_PROBLEM,AGENT_1));
fun PC n=(Mark.Student'END n=(STUDENT_1));
```

In the following the code for the formulae is shown. Using the atomic propositions previously shown, $formula1$ is used to verify that whenever it is possible to have *PA* evaluated to true, then eventually *PB* is also evaluated to true. The same reasoning holds for $formula2$.

```
val formula1 = AND(POS(PA),(EV(PB)));
val formula2 = AND(POS(PB),(EV(PC)));
```

The atomic propositions and formulae are similar for the verification of all plans. The major difference is related to the marking of the place to predicate, and how many predicates are necessary to prove the current plan. Therefore, it is not necessary to list all atomic propositions and formulae for all plan verification. The code for the problem solver plan verification can be used as an example.

Verification Guideline 5 – Model Checking. The last step is to perform model checking using the model and the atomic propositions and formulae specified in the Guide-

line 4 above. In the following the piece of code that implements this step is shown. The *eval_node* is used to evaluate the truth value for the formulae. Therefore, if both formulae are evaluated to true then the designer receives a message informing that the plan has been validated. Otherwise an error message is shown.

```
bReturn1 := eval_node formula1 InitNode;
bReturn2 := eval_node formula2 InitNode;
if ((!bReturn1 = true) andalso (!bReturn2 = true))
    then DSUI_UserAckMessage("The plan has been validated")
    else DSUI_UserAckMessage("There is a problem with this plan");
```

For this example the verification for one plan is detailed. Nevertheless, it is important to point out that the same reasoning strategy can be applied for every plan. Therefore, using theses guidelines it is possible to specify and verify all the plans for every activity of the agents.

6 Related Work

Many researchers have applied Petri nets to the multiagent domain. In [22] a Colored Petri net for a multiagent application, in order to analyze agent social behaviors, is presented. Agent cooperation and coordination are analyzed using colored petri nets in [23, 24]. An agent oriented programming extension for Colored Petri nets is presented in [25]. In [26], Recursive Petri Nets are used to model multiagent plans.

In the context of multiagent planning, the model checking approach has been used to verify plans of the agents based on various formalisms. In [27] it is proposed an OBDD based planning framework for multiagent and nondeterministic domains. In [28] is presented an approach for modelling and verifying multiagent behaviors using Predicate/Transition Nets. In [29] G-net is extended for modelling inheritance of agent classes in multiagent systems, which provides a clean interface between agents with asynchronous communication ability and supports formal reasoning.

There are other works and formalisms for multiagent plan modelling and verification. However, the approach proposed in this paper gathers various tools, such as MSC, simulation, and model checking, to provide a systematical way for modelling and verifying multiagent plans. Design CPN support makes the verification an automatic activity, representing an advantage if compared to Predicate/Transition works. Moreover, guidelines are generic and can be applied to any multiagent domain.

7 Final Remarks

In this paper it is introduced an approach to verify nondeterministic execution plans for multiagent systems using Hierarchical Colored Petri Nets. This approach is based on guidelines for the modelling and verification activities. These guidelines are used to fully implement the presented solution using available computational tools.

In order to illustrate the introduced approach, it has been applied to a multiagent intelligent tutoring system. Thus, the guidelines have been effectively applied to a realistic example. The presentation of the case study was kept simple enough to make the

concepts and the application of the guidelines clear. Moreover, based on the example, the applicability of the approach has been validated taking into account scalability and complexity issues. The scalability issue is implicitly treated, since the guidelines introduced are not coupled with the number of agents and the types of architectures that can be defined. Also, the approach promotes an effective way to manage the state explosion problem due to the fact that multiagent plans are verified based on local plans for types of architectures.

As future work automatic generation of plans is currently being investigated. This is important because for multiagent systems it can be difficult or even impossible to predict the behavior of all the agents in advance. Moreover, the approach discussed in this work is being applied to other domains such as, for example, sensor networks and manufacturing planning.

Acknowledgements

The research reported is this chapter is partially supported by grants 305110/2002-0 and 200365/2004-5 from the Brazilian National Research Council (CNPq), a scholarship from CNPq for the first author and a scholarship from CAPES for the second author.

References

1. Jennings, N.R.: An Agent-based Approach for Building Complex Software Systems. Commun. ACM **44** (2001) 35–41
2. Ball, D., Wyeth, G.: Multi-Robot Control in Highly Dynamic, Competitive Environments. In: RoboCup 2003: Robot Soccer World Cup VI. Volume 3020 of Lecture Notes in Computer Science. Springer-Verlag (2003) 385–396
3. Fernández, D.C., López, J.M.M., Millán, D.B.: A Multiagent Approach for Electronic Travel Planning. In: Proceedings of 2nd International Workshop Agent-Oriented Information Systems/CAiSE'00, Stockholm, Sweden (2000)
4. Henoch, J., Ulrich, H.: Agent-Based Simulation Platform for Evaluating Management Concepts. In: Proceedings of the 4th International Eurosim 2001 Congress, Delft, Netherlands, TUDelft Press (2001) 1–6
5. Kornienko, S., Kornienko, O., Levi, P.: Flexible Manufacturing Process Planning based on the Multi-agent Technology. In: Proceedings of 21st the IASTED International Conference on Applied Informatics, Innsbruck, Austria, ACTA Press (2003) 87–92
6. Upal, M.A., Fung, F.: Dynamic Plan Evaluation for Military Logistics. In: Proceedings of the Seventh International Conference on Artificial Intelligence and Soft Computing, ACTA Press (2003) 87–92
7. Xuan, P., Lesser, V.: Using Agent Commitments as Planning Contexts. International Journal on Cooperative Information Systems (2003, under review)
8. Bowling, M.H., Jensen, R.M., Veloso, M.M.: Multiagent Planning in the Presence of Multiple Goals. In: Intelligent Planning. Intelligent Series. Wiley (2004, to appear)
9. Pistore, M., Traverso, P.: Planning as Model Checking for Extended Goals in Non-deterministic Domains. In: IJCAI. (2001) 479–486
10. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press, Cambridge, Massachusetts (1999)
11. Giunchiglia, F., Traverso, P.: Planning as Model Checking. In: ECP. (1999) 1–20

12. Cimatti, A., Roveri, M.: Conformant Planning via Symbolic Model Checking. In Journal of Artificial Intelligence Research (JAIR) (2003) 305–338
13. Harel, D., Kugler, H.: Synthesizing State-Based Object Systems from LSC Specifications. International Journal of Foundations of Computer Science **13** (2002) 5–51
14. Jensen, K.: Coloured Petri Nets: Basic Concepts, Analysis, Methods and Practical Use. EACTS – Monographs on Theoretical Computer Science. Springer-Verlag (1992)
15. Jensen, K.: Coloured Petri Nets – Basic Concepts, Analysis Methods and Practical Use. Volume 2. Springer-Verlag (1997)
16. Weiss, G., ed.: Multiagent Systems – A Modern Approach to Distributed Artificial Intelligence. MIT Press (1999)
17. Jensen, K., al, e.: Design/CPN 4.0. Meta Software Corporation and Department of Computer Science, University of Aarhus, Denmark. (1999) On-line version: http://www.daimi.aau.dk/designCPN/.
18. Christensen, S., Mortensen, K.H.: Design/CPN ASK-CTL Manual, University of Aarhus. 0.9 edn. (1996)
19. Costa, E.B., Lopes, M.A., Ferneda, E.: MATHEMA: A Learning Environment Based on a Multi-Agent Architecture. In Wainer, J., Carvalho, A., eds.: Proceedings of the 12th Brazilian Symposium on Artificial Intelligence. Volume 991 of Lecture Notes in Artificial Intelligence., Campinas, Brasil, Springer-Verlag (1995) 141–150
20. Costa, E., Almeida, H.O., Lima, E.F.A., Filho, R.R.G.N., Silva, K.S., Assunção, F.M.: A Cooperative Intelligent Tutoring System: The case of Musical Harmony domain. In Coello, C., Albornoz, A., Sucar, L., Battistuti, O., eds.: Proceedings of 2nd Mexican International Conference on Artificial Intelligence – MICAI'02. Volume 2313 of Lecture Notes in Artificial Intelligence., Mérida, Yucatán, México, Springer Verlag (2002) 367–376
21. Silva, L.D., Almeida, H.O., Perkusich, A., Costa, E.B.: Modelling and Analysis of a Multi-Agent Intellgent Tutoring System Based on Coloured Petri Nets. In: 1st ACIS International Conference on Software Engineering Research and Applications (SERA'03). Volume 1., San Francisco, EUA, Mt. Pleasant: International Association for Computer and Information Sciences (ACIS) (2003) 276–281
22. Weyns, D., Holvoet, T.: A Coloured Petri Net for a Multi Agent Application. In Moldt, D., ed.: Proc. of the Second International Workshop on Modelling of Objects, Components, and Agents (MOCA'02), Aarhus, Denmark (2002) 121–140
23. Fiorino, H., Tessier, C.: Agent Cooperation: a Petri Net based Model. In: Proceedings of ICMAS'98. (1998) 4–7
24. Miranda, M., Perkusich, A.: Modeling and Analysis of a Multi-Agent System Using Colored Petri Nets. In: Proc. of Workshop on Applications of Petri Nets to Intelligent System Development, Williamsburg, Virginia, USA (1999) 87–99
25. Moldt, D., Wienberg, F.: Multi-Agent-Systems based on Coloured Petri Nets. In: Proceedings of the 18th International Conference on Application and Theory of Petri Nets, Springer-Verlag (1997) 82–101
26. Seghrouchni, A.E.F., Haddad, S.: A recursive model for distributed planning. In Lesser, V., ed.: Proceedings of the First International Conference on Multi–Agent Systems, MIT Press (1995)
27. Jensen, R., Veloso, M.: OBDD-based Universal Planning for Multiple Synchronized Agents in Non-Deterministic Domains. In: Proceedings of the Fifth International Conference on Artificial Intelligence Planning Systems, Breckenridge, CO (2000) 167–176
28. Xu, D., Volz, R., Ioerger, T., Yen, J.: Modeling and Verifying Multi-agent Behaviors using Predicate/Transition Nets. In: Proceedings of the 14th international conference on Software engineering and knowledge engineering, ACM Press (2002) 193–200
29. Xu, H., Shatz, S.M.: A Framework for Modeling Agent-Oriented Software. In: Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS). (2001)

Specification of Role-Based Interactions Components in Multi-agent Systems

Nabil Hameurlain[1] and Christophe Sibertin-Blanc[2]

[1] LIUPPA Laboratory, Avenue de l'Université, BP 1155, 64013 Pau, France
nabil.hameurlain@univ-pau.fr
http://www.univ-pau.fr/~hameur
[2] IRIT Laboratory, University Toulouse I, Place Anatole France31042 Toulouse, France
sibertin@univ-tlse1.fr

Abstract. Roles are an important concept used for different purposes as the modeling of the organizational structure of multi-agent systems, the modeling of protocols, and as basic building blocks for defining the behavior of agents. Modeling interactions by roles brings several advantages, the most important of which is the separation of concerns by distinguishing the agent-level and system-level with regard to interaction. However, in open MASs, the composition of independently developed roles can lead to unexpected emergent interaction among agents. This paper identifies requirements for modeling role-based interactions, and presents a formal specification model of roles for complex interactions. Our approach aims to integrate specification and verification of roles into the Component Based Development approach. An interaction protocol example is given to illustrate our formal framework.

1 Introduction

The Multi-Agent System (MAS) paradigm is one of the most promising approaches to create open and dynamic systems, where heterogeneous entities are naturally represented as interacting autonomous agents, which can enter or leave the system at will. Interaction among autonomous agents is fundamental to the dynamic of multi-agent systems. Agents belonging to a same application need to interact and coordinate their activity to carry out their common global goal, whereas agents belonging to different applications, as in an open scenario, also need to interact, for instance to compete for a resource.

If these interactions are uncoordinated, there is no chance that they lead to the achievement of the common goal, and the role concept is just the one that relates the interactions performed by agents and the objectives of the system. A role is a specific contribution to the system that realizes a part of the global goal, and it determines how this sub-goal may or must be achieved. Thus, roles are basic buildings blocks for defining the organization of multi-agent systems, together with the behavior of agents and the requirements on their interactions [18]. Modeling interactions by roles allows a separation of concerns by distinguishing the agent-level and system-level concerns with regard to interaction. An important characteristic of real-world agent systems is that an agent may have to change the role it plays over time. If some flexibility constraints require some variety of these roles, agents have to adapt their architecture and functionality as they adopt new roles. These additional capabilities must be dynami-

R. Choren et al. (Eds.): SELMAS 2004, LNCS 3390, pp. 180–197, 2005.
© Springer-Verlag Berlin Heidelberg 2005

cally acquired because only a few roles can be hard-coded into an agent. As a matter of fact, this dynamic acquisition is the only possibility in open system where agents enter and leave at will. While designing the overall organization of a system, it is valuable to reuse roles previously defined for similar applications, especially when the structure of interaction is complex. To this end, roles must be specified in an appropriate way, since the composition of independently developed roles can lead to the emergence of unexpected interaction among the agents.

On the other hand, Component Based Development (CBD) [26] promises to facilitate the construction of large-scale applications by supporting the composition of simple building blocks into complex applications. It is one of the most important among the recent technical initiatives in software engineering. In CBD, software systems are built by assembling components already developed and prepared for integration. Therefore, the specification of components is useful to both components users and components developers. The specification provides a definition of the component's interface and it must be precise and complete for users; for developers, the specification of a component also provides an abstract definition of its internal structure. The verification of such a well-established specification is needed for a safe composition of systems from components. Verification and CBD are synergistic: CBD introduces compositional structures, and composition rules to build systems, whereas specification along with verification enable the effective development of reliable component-based software systems.

It appears that the facilities brought by the CBD approach fit well the issues raised by the use of roles in MASs, and this paper makes a proposal in this way. It presents the RICO (Role-based Interactions COmponents) model for specifying complex interactions based on roles in open MAS. Although the concept of role has been exploited in several approaches [1, 2, 3, 4, 5, 7, 8, 18, 28] in the development of agent-based applications, no consensus has been reached about what is a role and how it should be specified and implemented. RICO proposes a specific definition of role, which is not in contrast with the approaches mentioned above, but is quite simple and can be exploited in specifications, validations and implementations. In RICO, a role includes a set of *interface elements* (either attributes or operations, which are the provided and required features necessary to accomplish the role's tasks), a *behavior* (interface elements semantics), and *properties* (proved to be satisfied by the behavior). When an agent intends to take a role, it creates a new component (i.e. an instance of the component type corresponding to this role) and this role-component is linked to its base-agent. Then, the role is enacted by the role-component and it interacts with the role-components of the other agents.

This paper focuses on the integration of specification and verification of roles into the Component Based Development. Section 2 defines requirements for modeling role-based interactions as components together with the RICO (Role-based Interactions COmponents) specification model for complex interactions based on roles. Section 3 presents the formal specification language COO (CoOperative Objects) [23], together with SYROCO [24] (an acronym for SYstème Réparti d'Objets CoOpératifs), an environment that implements COOs. In section 4 we map the proposed RICO specification model to the COO formalism, and specify properties of role-components. An example of interaction protocol is studied to illustrate our approach. We present related approaches in section 5 before to conclude in section 6.

2 Specifying Role-Based Interactions as Components

In the following, first we overview the specifications of software components in Component Based Software Engineering, and then we present the RICO model for specifying agent roles in open multi-agent systems, which is a template that can be instantiated on various concrete computation model. The main objective is to use the CBD approach for specifying role-based interactions as reusable components [10], and which can be combined together by matching their respective needs and services.

2.1 Specification of Software Components

Component specification is an important issue in CBD. Although this problem has been addressed from the very beginning of the development of component models, it remains one of the challenging problems of Component Based Software Engineering.

Up to now, the specifications of components used in practical software development are limited to syntactic specifications. These specifications include the specifications used with technologies such OMG's CORBA [20] or Sun's JavaBeans [25]. The first of these uses different dialects of IDL (Interface Definition Language) [9], whereas the second uses the Java programming language to specify component interfaces. Therefore, the specification of a component consists of a precise definition of the component's operations and context dependencies, and an essential feature of most component specification techniques is the independence of interfaces from the component implementations.

An important aspect of interface specifications is related to the principle of substitution of components. A component can be substituted if the new component implements at least the same interfaces as the older component. For substitution of components to be safe, however, several techniques try to provide semantics specifications, and most of them use UML [19] and its Object Constraint Language (OCL) [27]. Thus a component implements a set of interfaces, each of which consists of a set of operations. In addition, a set of preconditions and postconditions is associated with each operation, which often depend on the state maintained by the component [17]. Nevertheless this is in general insufficient for the re-usability and extendibility of components. Instead, the semantics or behavior of a software component has to be included in its specification, and the safe substitution of components needs to compare their behaviors [10].

2.2 RICO Specification Model

RICO (Role-based Interactions COmponents) is a role-based interactions abstract model for the specification of roles as components in agent-based applications. The main motivation for modeling role-based interactions as components is to capture interaction patterns that:

- feature well-defined and proved properties,
- may be composed the ones with the others so that the resulting behavior allows to realize an intended goal,
- may be dynamically linked to agent and dissociated when this is no longer necessary,

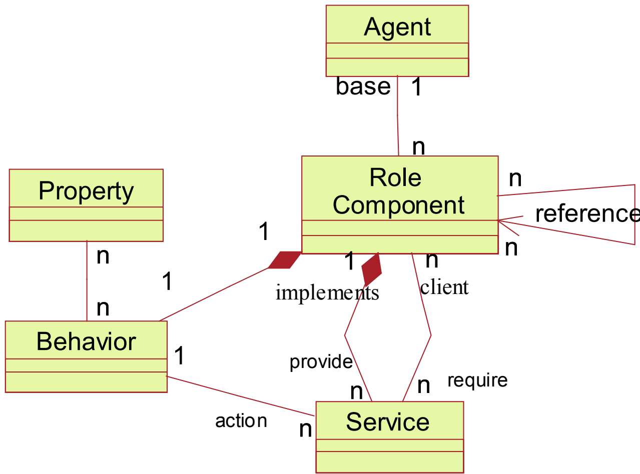


Fig. 1. UML metamodel of the concepts used in RICO specification model.

- enable a separation of concerns by distinguishing the agent-level and system-level with regard to interaction.

The concept of role has been used in several multiagent development methodologies for modelling and analysing complex system applications. Although there are several role definitions, in all approaches roles are used to identify some task, behaviour, responsibility or operation that should be performed within the system [3]. In RICO, a role is considered as a component providing a set of interface elements (either attributes or operations, which are provided or required features necessary to accomplish the role's tasks), a behavior (interface elements semantics), and properties (proved to be satisfied by the behavior). Figure 1 is a UML class diagram showing the concepts used in RICO and the relationships between them. This UML metamodel specifies that a role-component requires and provides some services, which must be implemented by others role-components, and a service is implemented by one role-component. This independence of services from the role component implementations is an essential feature of RICO specification model according to the CBD approach. Definition 1 gives the explicit definition of the concepts used in RICO. This model is a generic representation of the relationships between these concepts, since in practice, the expression of these relationships varies from one specification technique to another, that is, one can distinguish between object-oriented specifications and procedural specifications.

In RICO, agents may take up one or several roles simultaneously, and an agent can assume the same role several times, in the same conversation or in distinct ones. For the simplicity, and in order to avoid the conflict access to the agent's resources (for instance agent's public attributes), we assume that only one role is active at each moment in time, and this later is under the agent's decision [3]. Since we are interested in specifications and verification of roles, we assume that the relationship between agent and role is static, that is agents take up roles statically and not dynamically.

Definition 1. A Role Component for a role R is a n-tuple RC = (Ag, Ref, Serv, Behav, Prop), where:

1. *Ag* is the identity of the *base* agent to which RC is linked: Ag has created RC, and RC enacts the role R on its behalf and under its control.
2. *Ref* is a list of role component identities, the role components in the system that RC knows and with which it may interact.
3. *Serv* is the interface of RC, the set of public features through which it interacts with the role components registered in Ref. These features are either *attributes* or *operations*, *methods* according to the standard object-oriented denomination. In addition Serv' features are either provided or required. *Provided* features are maintained and operated by RC itself at the disposal of other role components; provided attributes may be read and provided operations may be called. *Required* features are features provided by other role components and used by RC; the proper behavior of RC needs these attributes and operations and thus depends on their proper behavior. Notice that the interface of a role component forms the basis for its interaction with the environment (agent holding that role and other role components).
4. *Behav* defines the behavior of RC with regard to the other role components of the system. It describes the life-cycle of RC and the sequences of observable actions supported by RC as either the caller of a required operation or the callee of a provided operation; defining Behav in this way assumes that there is no constraint on the access to public attributes; if not, the availability of these attributes must also be specified in any manner. Serv together with Behav may be considered as the functional specification of RC, Behav being a language defined on the operations of Serv, and concerns the Serv's elements by capturing their precise behavior. For instance, the semantics may be specified by using pre- and post-condition associations, describing namely the life-cycle of the role component: sequence, synchronization, and concurrency of operations. Notice that the definition of a role component may also be completed with the mention of private attributes and operations and Behav may includes unobservable actions (private operations), and therefore encapsulates the implementation of the component.
5. *Prop* is a set of behavioral properties that are proved to be satisfied by Behav, so that components requiring the services provided by RC can trust in their fulfillment. Safety and liveness properties are of specific interest [15]: safety properties are invariants that states "nothing bad happens". In contrast, liveness properties state "something good happens". They are a functional specification of RC that is more abstract than Behav, more declarative in that they are just statements of properties that are guaranteed to be fulfilled by RC in any case. Role's functional properties are useful for selecting a role component and for assessing its suitability, usability or reuse, relative to a given application.

The execution semantics of the Role Components is defined as follows: when an agent executes, if the agent has taken up one or several roles, then at any moment exactly one RC executes (interleaving semantics). Role Components interacts with each other through the call of provided operations. Messages input or output by RC are consumed or generated by Behav through the interface Serv of the Role Component. Role Components allow a proper means for modelling agents and complex interactions, since:

- Roles components are *reactive*, *proactive*, and *autonomous*: through their interface, reactivity is dealt with by operations and services provided, proactiveness is dealt with by services required, and finally a role component execute autonomously according to its behavior Behav, which can be non deterministic. Thus role components can be used as agent-building blocks.
- Considering role components as the active members of protocols facilitates the modeling of the behavior of complex interaction protocols, especially open and concurrent ones. Thus, an agent can play one or more roles at the same time in different conversations (protocols occurrences), and each participation is managed by a specific role component.

In this paper we are interested in an object-oriented specification and implementation of the RICO model. We focus on the specification of these role components, and their implementation by a concurrent formal object-oriented language, the Cooperative Object (COO) formalism. With respect to the specification and design, we have a similar view as, for example, the Gaia methodology [28], and the main focus of this paper is the specification, verification and the implementation of roles.

3 The CoOperative Objects Formalism

In this section we present the COO formalism, a fully concurrent object-oriented formal specifications language, and its implementation SYROCO, together with an example of interaction protocols.

3.1 COO Definition

CoOperative Objects (COO) [23] is a formal specifications language allowing to model a system as a collection of active objects cooperating through an asynchronous request / reply protocol. Each object, being an instance of its COO class, has an identity usable as a reference, and may be dynamically created and deleted.

The structure of a COO includes a set of *attributes*, a set of *operations*, a control structure net called its *OBCS* (Object Control Structure), and a set of *services* supported by this OBCS. The definition of a COO class is divided into two parts, (see an example in section 3.2): the *specification* part concerns the public provided items composing the interface, while the *implementation* part includes the private items, notably the OBCS, a Petri net with objects [16] defining its control structure. Services are public provided methods – with typed in- and out-parameters – and service requests are processed according to the state of the OBCS.

The OBCS of a COO is a high-level Petri net made of transitions, places and arcs, each of them labeled with inscriptions referring to the processed data. Places are state variables whose values (multi-sets of tokens referred to as their marking) determine the current state of the object. Transitions correspond to actions that the object is able to perform, and the occurrence of a transition produces a change in the net's marking. Arcs from places to a transition determine the enabling condition of the transition, while arcs from a transition to places determine the result of its occurrence; the variables labeling arcs surrounding a transition are formal parameters defining how a transition occurrence moves tokens from input to output places. In addition, the value

of the tokens linked to these variables at an occurrence of the transition, can be accounted for by means of a guard and processed by means of a piece of code, the transition's *action*. The action of a transition – written in any sequential object-oriented programming language – has access to the attributes and operations of the object as well as to the public attributes and services of other objects.

Some arcs feature a particular shape with the following semantics:

- Inhibitor arc, with a circle on the transition side: the transition is enabled only if the place contains no token (transition t2 in figure 2);
- Place-to-transition clearing arc, with a double arrowhead: each occurrence of the transition removes all the tokens from the input place. An integer value labeling the arc indicates the minimum number of tokens necessary to enable the transition (transition t1 in figure 2).

In this paper, we consider services just with a message sending semantics instead of the full asynchronous request–reply semantics. Each provided service is associated with transitions which process the receipt of requests for that service (*accept-transition*), shown by a pending input arc labeled by the service's in-parameters. Thus, a request for a service is processed by occurrences of transitions in the course of the execution of the OBCS, and it is processed in different way according to the accept-transition that takes the request-token. From the client side, a request for a required service is issued by a *request-transition*. It is distinguished by a pending output arc labeled by the service's actual parameter and its occurrence gives a token to the accept-transitions of the requested service.

The activity of a COO instance consists in processing the calls for its provided operations upon request, and in executing its OBCS as a background task: while transitions are enabled under the current marking, it selects one of them and makes it occur.

The global behavior of a COO system results from the concurrent execution of its constitutive objects. Usually, we need to compose their OBCS for the analysis; this composition is in asynchronous way according to the message sending semantics of service rendering: given a client COO and a server COO, the composition of their OBCS consists in connecting, through communication places, the request-transitions and the accept-transitions for the same service: each provided service goes with an *entry-place* for receiving the requests. Then, a Client is connected with the service provider through this communication place by an arc from each request-transition towards the suitable entry-place and an arc from the suitable entry-place towards each accept-transition of the service.

The COO formalism is supported by SYROCO [24], an environment that makes it possible to edit COO class and to generate a C++ class for each COO class, for sequential computing environments (interleaving semantics among objects), for environments supporting threads and also for distributed computing environments compliant with CORBA (true parallelism semantics among objects). SYROCO offers symbolic debugging facilities allowing the designer to examine the state of the OBCS [24], that is the sate of the object (history of transitions occurrence, the previous and the next transition occurrence, value of tokens…). This debugger does not deal with the code of actions, but with the behavior and cooperation among Objects. Each COO has it own debugger, and it is possible to call the debugger of any object from the debugger of another one.

3.2 An Application Example

To illustrate the COO formalism, we will study an example of interaction protocols, the fish-market auction protocol. In [11], authors show how to model this protocol by means of a COO class. In this paper, we show how to design the COO classes that model the two roles, that is, the Vendor and Buyer roles. In any *conversation* following the rules of this protocol, we have a single vendor, and a number of potential buyers, the bidders. The vendor has a bucket of fish to sell for an initial price. A buyer can make a bid to signal its interest. If no (or more than one) buyer is interested, the vendor announces a new lower (or higher) price. When one and only one buyer is interested, the vendor attributes fish to that bidder. Once the bucket of fish is attributed, the vendor gives the fish and receives the payment, while the buyer pays the price and receives the fish.

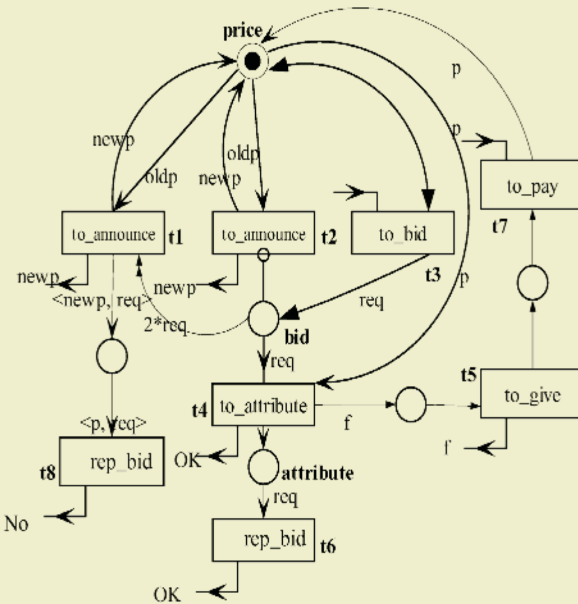
First, let us consider the fm_Vendor class; figure 2 gives its specification and implementation as a CoOperative Object class. The behavior (Control structure) of this class is as follows: under the initial marking (one token in the `price` place), only the `t2` and `t3` transitions are enabled and may occur. They are respectively the request-transition and accept-transition of the `to_announce` and `to_bid` services, and the vendor may process only these services. The acceptance of a `to_bid` service (by transition `t3`) produces new token into the `bid` place; the transition `t3` remains enabled as long as there is a token in the `price` place, that is until an occurrence of the `t4` transition caused by a request for the `to_attribute` service. This service is accepted if there is exactly one token in the `bid` place, that is there is a single current bidder. The occurrence of `t4` returns `OK` to the buyer, and enables both the request-transition `t5` of the `to_give`, and the request-transition `t6` of the `rep_bid` services. The occurrence of the `t6` transition returns `Ok` to the buyer, and the occurrence of `to_give` service enables the accept-transition `t7` of the `to_pay` service. The final state of the conversation is reached since the marking of the OBCS becomes the initial one, that is one token in the `price` place.

In figure 3, we give the fm_Buyer class. The behavior of this class is as follows: under the initial marking (one token in the `portfolio` place), only the `t1` transition is enabled and may occur. It is an accept-transition of the `to_announce` service. The acceptance of the `to_announce` service produces an occurrence of the `t6` transition that requests the `to_bid` service. Then, transition `t2` accepts the rep_bid service from the vendor and receives the Res reply. If the value of `Res` is true, it produces a new token into the `attribute` place, and enables the accept-transition of the `to_attribute` service; thus the occurrence of the `to_attribute` service enables `t4`, a request-transition of the `to_pay` service. The rendering of the `to_pay` service enables the accept-transition `t5` of the `to_give` service. Otherwise, if the value of `RES` is false it produces a token into the `announce` place, and the buyer should wait for a new announce from the vendor. The final state of the conversation is reached since the marking of the OBCS becomes the initial one, that is one token in the `portfolio` place, and one token in the `announce` place.

```
class fm_Vendor specification;
attributes
bidders: list of agent*;                 //list of bidder agents
vendor: agent*;                          //the creator agent
operations          //the C++ code of operations is not shown
Bidder_ident():agent* is <..>;         //identity of final bidder
Bidder_num():integer is <...>;        //current number of bidders
Current_price(): Currency is <...>;    //current price of auction
~Vendor() is <...>;
Vendor(vendor:agent*, p:Currency): Vendor * is <...>;

Services
             // service provided
to_bid();                           //receive a bid
to_pay(p: Currency): Status;        //receive a payment
             // services required
to_announce(newp: Currency);        //send the new price
to_attribute();
to_give(f: fish);
rep_bid(reply: Boolean);            //reply to a received bid
end.

class fm_Vendor implementation;
attributes
current-price: Currency;
OBCS
```




```
end.
```

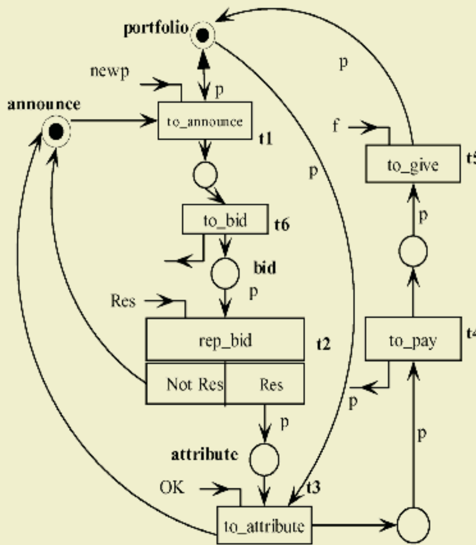
Fig. 2. The Vendor role in the fish-market protocol as a COO class, fm_Vendor.

```
class fm_Buyer specification;
attributes
bidder: agent*;                           //the creator agent
vendor: agent*;                           //the vendor agent
other_bidders: list of agent*;

operations                      //C++ code of operations not shown
Vendor-ident (): Agent* is <..>;          //identity of vendor
Current-price(): Currency is <...>;    //current price of auction
~Buyer () is <...>;
Buyer (bidder: agent*, portfolio: Currency): Buyer* is <...>;

services
             // service required
to_bid();
to_pay(p: Currency): Status;
             // services provided
to_announce(newp: Currency);
to_attribute();
to_give(f: fish);
rep_bid(reply: Boolean);
end.

class fm_Buyer implementation;
attributes
portfolio: Currency;
current-price: Currency;
OBCS
```




```
end.
```

Fig. 3. The Buyer role in the fish-market protocol as a COO class, fm_Buyer.

4 Implementation of RICO Model as COO

In this section we show how to map the proposed RICO specification model to the COO formalism, together with some characteristic properties, and give some safety and liveness property of role components and their verification.

4.1 Mapping RICO Model to COO

The mapping of RICO model to COO consists in modeling Role Components as COO classes, both at the *instance* and *type* levels. Referring to definition 1 of a Role Component RC, we have:

- Ag and Ref are registered in attributes, either in the specification or in the implementation according to visibility considerations.
- The provided features are exactly the element declared in the specification part; concerning the required features, services are the ones attached to the request-transitions of the OBCS such as e.g. the services `to_bid` and `to_pay` of transitions `t6` and `t4` in Figure 3, while attributes and operations can be explicitly listed as comment.
- The behavior of RC is defined by its OBCS that rigorously determines (1) when a reception of a request for a provided service can be taken into account and processed by an accept-transition occurrence and (2) when a request for a required service is issued by a request-transition occurrence. The capabilities and the needs wrt to message receptions and sending are thus formally expressed by the RC's OBCS. Other interactions among role components are never constraining and thus have no effect on their respective behaviors, that is: public attributes are continuously readable and calls for operations are synchronously processed upon request.
- The behavioral properties are properties of the OBCS and their technical statement may follow a variety of expressions, some are given below.

As mentioned by Kristensen [14], the concept of roles in object-oriented modeling, should support a set of characteristic properties; the specification and the implementation of RICO model as COO support these properties as follows:

- Visibility/ Dependency: visibility is supported by distinguishing a specification part concerning public items of the interface (operations and services), and implementation part concerning private items, notably the OBCS. Dependency is supported by the fact that the existence of a Role Component depends on that of the agent playing this role.
- Identity/ Dynamicity: supported by the fact that each role component as COO, being an instance of its COO class, has its own identity that can be used as reference, and may be dynamically created and deleted by the agent playing this role.
- Multiplicity/ Abstractivity: supported by the fact that the role components are mapped to COO classes, distinguishing between Role Components on *instance* and *type* level. Thus, several instances of role components may exist for a role at the same time.

Clearly the COO language is not the only way to implement the RICO model, RICO can be supported by any language allowing to explicitly:

- identify and characterize elementary interactions (for Serv);
- describe formally a control structure for Behav;
- have operational semantics in order to deduce more easily an executable implementation from the specifications;
- have compositional semantics in order to deduce emergent interaction among Role Components.

For instance, the type of automatons shown in figure 4 and 5 (section 4.2), could be used to describe the behavior of fm_vendor and fm_buyer classes. The main advantages to use petri nets is that they are completely compositional, that is the composition of the OBCSs (for different roles components) is also an OBCS, even if role components enter and leave the conversations at will [23]; while the definition and the semantics of communicating automatons are more difficult than those of simple automatons, and structural modification of the system (e.g. create, insert and delete an automaton) are more difficult to be taken into account. This mechanism of composition is essential to verify properties related to emergent behaviour when independently developed components roles are put in interaction: the properties of the system are deduced from the properties of its components.

4.2 Safety and Liveness Properties of Role Components

Since the life-cycle of role components is modeled by means of Petri nets, two kinds of properties should be verified [16]: structural property which are related to Petri nets topology. These kinds of properties help designer to build correct specification of the role, independent of the number of agents, and resources; and behavioral properties, that depend on the fixed initial state, and concern qualitative behavior. In this paper, we are interested in safety and liveness behavioral properties of roles and their verification.

Usually, it is possible to generate the reachability graph of the Petri net [16], using tools such as INA [22]. The reachability graph shows all the reachable states and all the sequences of transitions occurrences that may be performed. If the reachability graph is infinite, due to the infinity of a domain value, or to the fact that an action can be repeated again and again (for instance, transition `t3` of `to_bid` service in the Vendor component, figure 2), the covering graph is generated instead; it is finite, and allows the analysis of some safety and liveness properties of the net too.

Safety Properties. *Safety* properties of roles are requirements on finite execution. That is, statements of the form “nothing bad happens”. For instance, a specific attribute in a role component is always initialized before this role is taken by an agent. In the Fish-market protocol example, the identity and the initial price of the fish must be fixed by the agent taking the Vendor role (agent who initiates the protocol), that is attributes {vendor, current-price} are not null. These properties can be specified by means of predicates, expressed over the variables listed in the interface of RC. Safety properties express requirements which refer not only to several such status fields at once, but also to a history of states. For instance, using the covering graph of the fm_Vendor’s OBCS shown in Figure 4 (the symbols ! and ? are used to indicate respectively the required and provide services), we technically verify requirements that can be worded in the following way:

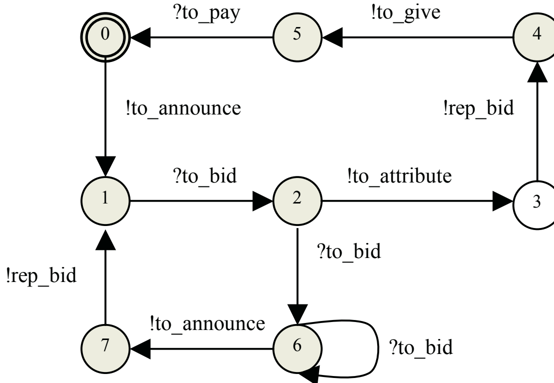


Fig. 4. The covering graph of the Vendor in the fish market protocol.

- "to_announce service may be performed from the initial state or, if, since its previous occurrence, no or more than one to_bid service has been performed".
- "to_attribute service may be performed exactly once when, since the previous to_announce occurrence, a single to_bid intervention has been performed.
- "to_give" and "to_pay" may be performed only once.

In addition to this classical behavioral analysis, the flexibility and the openness of SYROCO allow to add to the definition of a COO class new attributes to extend the structure of the tokens of any place of the OBCS, and to integrate new methods (functions) to be executed at each occurrence of transitions, namely when the interpreter of the OBCS removes tokens (from) or adds (into) the places. Besides, it is always possible to add, in the modeling of a COO class, transitions or places whose roles are not functional but only for supervising and detecting particular situations, and namely to be used to check safety properties of the Role Component .

Liveness Properties. As mentioned above, safety properties are a very powerful way to guarantee the correctness of a role by verifying that it never reaches an erroneous state. Sometimes this is not enough, and we need to claim that "something good eventually happens". This is the aim of the *liveness* properties. These subtle properties require checking for specific cycles in the reachability graph. So, a liveness property is violated if there is an infinite execution (trace) where progress is not guaranteed; usually, this means that some actions can be repeated infinitely, and the same states of the Role Component are visited again and again. In our example, the analysis of the fm_Vendor's OBCS, tells us that the initial state of the Vendor can be reproduced, and since the initial marking represents the state where there is no ongoing (active) conversation, this reversible property proves that every conversation can be eventually completed and finished. Besides, by exploring the reachability graph of the fm_Buyer's OBCS shown in Figure 5, we can verify requirements such as:

- "after a to_attribute intervention, to_pay intervention may be performed".
- "after a to_pay intervention, to_give intervention may be performed".

To further prove additional behavioral properties of roles, INA tool also provides some *state-based* model checking capabilities, and allows us to state properties in the

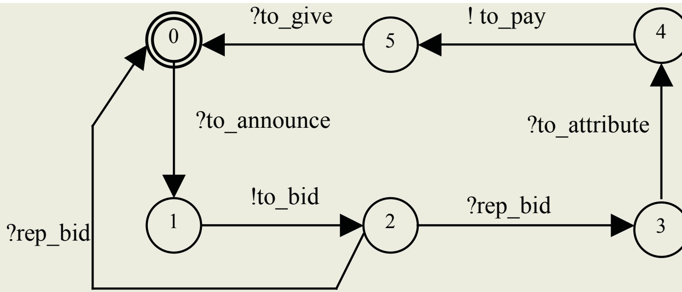


Fig. 5. The reachability graph of the Buyer in the fish market protocol.

form of CTL formulae [6]. These formulas are defined on the marking of places; so we can specify and verify some key temporal properties about roles on the whole reachability graph of the *Vendor--Buyer* OBCS, the Petri net with Objects obtained by the composition of the OBCS of the fm_Buyer and fm_Vendor classes according to the message sending semantics. For instance:

- *Mutual exclusion*, which is a safety property: for instance in the Vendor-component, no more than one buyer is selected to attribute the fish; it is expressed by the fact that it is impossible to mark both places `bid` and `attribute` at the same time. This property is expressed by:
  `EF(Vendor.bid & Vendor.attribute)`
  that results in `FALSE` if it is impossible to mark both places `announce` and `attribute` at the same time.
- *Concurrency* between role components, which is a liveness property: for instance there exists a path in the reachability graph of the *Vendor/Buyer_OBCS*, in which the `price` place (in fm_Vendor's OBCS), and the `bid` place (in fm_Buyer's OBCS) are marked at the same time, and then, `to_announce` and `to_bid` services, may be executed concurrently. This property is expressed by the holding of the following formula: `EF(Vendor.price & Buyer.bid).`

5 Related Work

There are many approaches and methodologies for the specification of roles (interactions) in multi-agents system. In recent years, roles formation, configuration among roles, and static semantics of roles have been proposed [2, 7]. [7] proposes a meta-model to define models of organizations, based on three concepts: agent, group, and role; agents belong to groups where they hold roles, and interactions take place only between agents member of the same group and according to their respective roles. Our approach is in the same line, since it is based on roles, and the agents that hold a role in the same conversation of a protocol constitute a group. However, our approach gives a formal and precise definition of the interaction patterns – protocols and roles – and groups are defined on the basis of conversations, i.e. occurrences of protocols. In [2], authors study the conditions under which an agent can enact a role and what it means for an agent to enact a role. They define possible relations between roles and

agents, and discuss functional changes that an agent must undergo when it enters an open agent system. This work completes our approach, and one can use the proposed relations as constraints interaction requirements that the agents that take up the role must meet. In [3], they argue for the importance of enactment/deactement of roles by agents in multiagent programming, in particular when dealing with open systems. This work study the dynamics of roles in terms of operations performed by agents; their formalization is conceptually based on the notion of cognitive agents, and therefore, we claim that it can be easily exploited in our specification and implementation of role components. In [28], Gaia methodology adopts an abstract, semiformal description to express the capabilities and expected behaviors of roles involved in protocols. This work is close to ours, since it is based on the organizational abstractions for analysis and design of complex and open interactions, but one possible limitation, is the formal specification, validation and namely the implementation of roles. This is due to the fact that, the life-cycle of roles in Gaia is only expressed by safety and liveness properties, and this methodology does not directly deals with formal analysis and implementation issues.

The Aspect Oriented Programming (AOP) approach has been exploited to implement the concept of roles in [12]. The author discusses the relevance of modeling roles for agents systems. In our approach, roles are considered as components for the interactions between agent's applications. Then, a Role can be selected and reused, considering not only its functionality but also its behavioral properties. In [5], the authors go beyond these AOP's considerations, and propose an interaction model based on the notion of XRole (XML Role), where notations based on XML are adopted to support the definition and the exploitation of roles at different phases of the application development. This work is close to ours, since it is based on the separation of concerns introduced by AOP, but it suffers from the lack of a formal semantics and as a consequence the possibility to make verification and validation. Thanks to their XML-based syntactic definition, XRole focuses on flexibility, openness, and adaptability of the roles, but not on their formal specification and verification.

Considering the specification and validation of complex interactions and open software systems, [13] proposes an extension of AUML for the modular design of interaction protocols by composing micro-protocols; the main contribution of this approach is to reduce the gap between informal specification of interaction and semi-formal one by using protocol diagrams (AUML sequence diagrams), a graphical language for designing protocols. Nevertheless, specification and verification of open interaction protocols remains non trivial process. In [4], authors develop a role concept for a modeling approach based on the UML and graph transformation systems. They also provide a run-time semantics for roles on concepts from the theory of graph transformation. This approach allows a convenient model for the concurrency, reactivity, and the autonomy of agents. Nevertheless, engineering issues raised related to the use of roles such as the validation and the verification of agent's behavior. In [1], the specification of the MAS is based on a hierarchy of components, defined in term of input/output and temporal constraints. The proposed framework is developed for specification and simulation of MAS. However, the approach has two drawbacks. First, with this approach it seems difficult to refine specifications to implementation language. Second, the verification technique is limited to model checking. [21] proposes a formal framework using the Z language, where initial units (schemas) of

specifications are refined in order to obtain a MAS specification. Nevertheless, this framework does not allow to specify temporal and reactive properties of MAS. In our model these aspects are specified by the behaviour and the properties of the Role Components. [8] proposes a multi-formalism based specification approach, including statecharts in Object-Z classes and proposes a formal framework approach for the prototyping and simulation of MAS. Although this approach allows specification and simulation, it has some limitations. Indeed, the Object-Z part of the specification is not yet executable, and only the analysis by simulation of the statechart specifications is possible.

6 Conclusion and Future Work

The aim of this paper is to show how to exploit the concept of role in engineering agent complex interactions (specification, verification, and implementation). Modeling interactions by roles gives several advantages, the most important of which is the separation of concerns by distinguishing the agent-level and system-level with regard to interaction.

Component Based Development (CBD) promises to facilitate the construction of large-scale applications by supporting the composition of simple building blocks into complex applications. Software systems are built by assembling components already developed and prepared for integration.

Starting from the above considerations, we have identified requirements for modeling roles-based interactions as components, and propose RICO (Role-based Interactions COmponents), a role-based interactions specification model whose aims is to specify roles in agent-based applications according to component based approach. Then, we have shown how RICO model can been specified and implemented by a concurrent formal object-oriented language, the CoOperative Object (COO) formalism, that enables the formal specification, analysis and validation of open and concurrent interactions. Finally, we have shown how to specify and check properties of a role component exploiting Petri nets theory: reachability graphs for behavioral analysis, and namely others tools such as INA tool for checking temporal properties.

The next step for this work is to exploit some directions. First, we want to adapt SYROCO environment in order to develop an infrastructure supporting RICO model for real size applications, and namely for open and concurrent interaction protocols. Our intention is to explore the coordination model based on Moderators of conversation presented in [11]. This model fits the organization-centered view of MAS as it strictly distinguishes the agent-level and the organization-level concerns with regard to interaction, and the main advantage of this approach is that it is quite simple, both to use and to implement. Second, we are exploring the formal specifications of the relationships between role components and organizational rules [28] such as compatibility and consistency between agents and roles, and interdependence of roles. Finally, we are going to consider non-functional properties in the specification of role components, such as performance, reliability, security, and environmental assumptions; the specification of non-functional properties is still an open area of research in Component Based Software Engineering, and we believe that it will have an impact on the future of software role components specification.

Acknowledgments

This work is funded by the STIC-CNRS Department, under the grant SUB/2003/076/DR16, in the context of C2ECL (Coordination et Contrôle de l'Execution de Composants Logiciels) action.

References

1. F. M. T. Brazier, B. Dunin Keplicz, N. Jennings, J. Treur, "Desire: Modelling Multi-agent Systems in a Compositional Formal Framework", International Journal of Cooperative Information Systems, 6:67-94, 1997.
2. M. Dastani, V. Dignum, F. Dignum, "Role Assignment in Open Agent Societies", AAMAS'03, ACM 2003.
3. M. Dastani, M. B. van Riemsdijk, J.Huslstijn, F. Dignum, J-J. Meyer, "Enacting and Deacting Roles in Agent Programming", AOSE'04.
4. R. Depke, R.Heckel, J.M.Kuster, "Roles in Agent-Oriented Modeling", International Journal of Software engineering and Knowledge engineering, vol 11, No. 3 (2001) 281-302.
5. G. Cabri, L. Leonardi, F. Zambonelli "BRAIN: a Framework for Flexible Role-based Interactions in Multi-agent Systems", Proceedings of CoopIS 2003, 2003.
6. E. M. Clarke, E.A. Emerson, A. P. Sistla, "Automatic Verification of finite-State Concurrent Systems using Temporal Logic Specifications", ACM Transactions on Programming Languages and Systems, Vol 8, N° 2, 1986, pp244-263.
7. J. Ferber, O. Gutknecht, "Aalaadin: A Meta-model for the Analysis and Design of Organizations in Multiagent system", ICMAS'98, 1998.
8. P. Gruer, V. Hilaire, A. Koukam, "Formal Specification and Verification of Multi-agent Systems", ICMAS'2000, IEEE, 2000.
9. M. Gudgin, "Essential IDL: Interface Design for COM", Reading, MA, Addison-Wesley, 2001.
10. N. Hameurlain, "Formal Semantics for Behavioural Substitutability of Agent Components: Application to Interaction Protocols", From Theory to Practice in Multi-agent Systems, LNAI 2296, Springer-Verlag, pp 131-140, 2002.
11. C. Hanachi, C. Sibertin-Blanc, "Protocol Moderators as Active Middle-Agents in Multi-Agent Systems", AAMAS, 8, 3, p. 131-164, Kluwer Academic Publishers, 2004.
12. E. A. Kendall, "Role Modelling for Agent Systems Analysis, Design and Implementation", IEEE Concurrency, 8(2): 34-41, April-June 2000.
13. J-L. Koning, M-P. Huget, J. Wei, X. Wang. Extended Modeling Languages for Interaction Protocol Design. AOSE'2001, Springer-Verlag, pp 93-100, 2001
14. B.B. Kristensen, "Object Oriented Modeling with Roles", in Proc. 2nd International Conference on Object-Oriented Information Systems (OOIS'95), pp 57-71, Springer .
15. Z. Manna, A. Pnueli, "Temporal Verification of Reactive Systems-Safety", Springer-Verlag, 1995.
16. T. Murata, "Petri Nets: Properties, Analysis and Applications", In *Proceedings of the IEEE*, Vol.77, No.4 pp.541-580, April, 1989.
17. B. Meyer, "Object-Oriented software Construction", Upper Saddle River, NJ, Prentice Hall, 1997.
18. J. Odell, H. V. .D . Parunak, S. Brueckner, J. Sauter, "Temporal Aspects of Dynamic Role Assignment", AOSE'03, Springer. 2003.
19. OMG, "OMG Unified Modeling Language specifications", Report V1.3, OMG, June 1999.
20. OMG, "The Common Object Request Broker: Architecture and Specifications", Report V2.4, OMG, 2000.
21. M. Luck, M. d'Inverno, "A formal Framework for Agency and Autonomy", ICMAS'95, AAAI Press/MIT Press, editor.

22. S. Roch, P. H. Starke, "INA: Integrated Net Analyzer, Version 2.2", Humboldt-Universitat of Berlin, April 1999.
23. C. Sibertin-Blanc, "CoOperative Objects: Principles, Use and Implementation", In Petri Nets and Object Orientation, G. Agha, F. De Cindio eds, LNCS 2001, Springer-Verlag. 2001.
24. C. Sibertin-Blanc et Al., "SYROCO: Reference Manual V7", University Toulouse1, Oct 1996, (C) 1995, 97, CNET and University Toulouse 1.
    Available at http://www.daimi.aau.dk/PetriNet/tools.
25. Sun Microsystems, "JavaBeans 1.01 Specification",
    Available at http://java.sun.com/beans.
26. C. Szyperski, "Component Software-Beyond Object-Oriented Programming", Addison-Wesley, 2002.
27. J. Warmer, A. Kleppe, "The Object Constraint Language", Reading, MA: Addison Wesley, 1999.
28. F. Zambonelli, N. Jennings, M. Wooldridge, "Developing Multiagent Systems: The Gaia Methodology", ACM Transactions on Software Engineering and Methodology, Vol 12, N° 3, July 2003, pp317-370.

The ANote Modeling Language for Agent-Oriented Specification

Ricardo Choren[1] and Carlos Lucena[2]

[1] Military Institute of Engineering, Department of Systems Engineering
Praça General Tibúrcio 80, Praia Vermelha, Rio de Janeiro / RJ, 22290-270, Brazil
choren@de9.ime.eb.br
[2] Pontifical Catholic University of Rio de Janeiro, Computer Science Department
Rua Marquês de São Vicente 225, Gávea, Rio de Janeiro / RJ, 22451-900, Brazil
lucena@inf.puc-rio.br

Abstract. Multi-agent systems are distributed systems of loosely coupled agents. Description and construction of these systems are eased by separating their structure from their dynamic behavior. ANote is a modeling language for multi-agent system analysis that supports this approach. It provides a notation language which supports multi-agent system analysis through its decomposition into structural and behavioral views. Each view is responsible for picturing an important aspect, while ignoring less important details. This paper describes the ANote notation language and its views. The notation language is described and illustrated by an example, an e-insurance system.

1 Introduction

Agent-based computing is rapidly emerging as a powerful technology for the development of complex distributed software systems, synthesizing contributions from many different research areas including artificial intelligence and software engineering [39]. To foster the creation of agent-based applications, new software engineering techniques are currently in progress. In this approach to software development, application programs are built of software agents. An agent is a module that embodies some goal, some actions – which operate to achieve these goals – and some high-level message interface, i.e. it is a software component that executes autonomously and communicate with their peers by exchanging messages in an expressive communication language [17].

Many platforms, frameworks, programming languages, modeling languages, methodologies, and so on, were developed to face the challenges and promises of the agent technology in recent years. However, compared to previous efforts in software engineering, such as in the object-oriented paradigm, the work in agent-oriented software engineering is still in its beginning [35]. In particular, we believe that a modeling language is an indispensable element to assist the agent-based software technology.

Although we believe that the object-oriented paradigm (e.g. UML-based approaches) is the most popular paradigm for software development, it has some flaws when it comes to agent-based application development [15, 16]. Software engineers who design and implement multi-agent systems are faced with concerns such as autonomy, interaction and adaptation that are not naturally supported by abstractions

R. Choren et al. (Eds.): SELMAS 2004, LNCS 3390, pp. 198–212, 2005.
© Springer-Verlag Berlin Heidelberg 2005

associated with object-oriented software engineering [15]. It turns out that the developer has to make several manipulations in the design to accommodate the agency properties to the object-oriented paradigm. This leads to a poor production of multi-agent systems, which are hard to maintain, evolve and reuse.

So, directly using an object-oriented modeling language, or an extension, is a limiting factor in an agent-based software environment. UML, for instance, is visual modeling language to develop and exchange meaningful models for Object Analysis and Design, as stated in the UML Specification Document [28]. A UML profile is not sufficient, because it would mean that an agent could be seen as a stereotyped or constrained object. A stereotype or a constraint does not change the nature of the item it affects, i.e. a stereotyped object is still an object.

UML would have to be changed to include notation for goals, agents, plans, organizations and agency properties, for example. To properly support agent-based modeling, it would be necessary to add new concepts and notations in the UML core meta-model. This addition would increase the language complexity thus making it more difficult to model an agent system.

Indeed, the definition of a conceptual meta-model is a key issue to encourage the use of a new paradigm. A conceptual meta-model is very important for problem understanding (conceptual modeling) and solution proposal (computational modeling) of any complex system development [12] since it appropriately describes the problem domain and its abstractions.

This paper proposes an approach that aims to specify a multi-agent system. This approach is based on the definition of an agent-based conceptual meta-model, and it uses integrated views to better specify the system's features. These views rely on the concepts defined in the meta-model and they classify the structural and dynamic aspects of a multi-agent system. Each view is modeled and documented using ANote. ANote is a notation language for visualizing, specifying, constructing, and documenting the artifacts of a multi-agent system specification. ANote provides a set of diagrams, each one modeling a different view of a multi-agent system. The approach is illustrated by an example, an e-insurance system, which has been analyzed using the ANote modeling language.

The remainder of the paper is structured as follows. Section 2 describes the ANote conceptual meta-model. Section 3 describes the ANote views. Section 4 explains the modeling language by showing a sample case study. A discussion of related work is presented in Section 5, while Section 6 summarizes the paper and offers directions for future work.

2 The ANote Conceptual Meta-model

The ANote modeling language was developed to offer a standard way to describe concepts related to the multi-agent system modeling process. In the present section we present the ANote meta-model for constructing a multi-agent system design. This meta-model provides a schema of the concepts, their relations, and some constraints that build a multi-agent solution.

The main concepts defined in the ANote meta-model (Fig. 1) are:

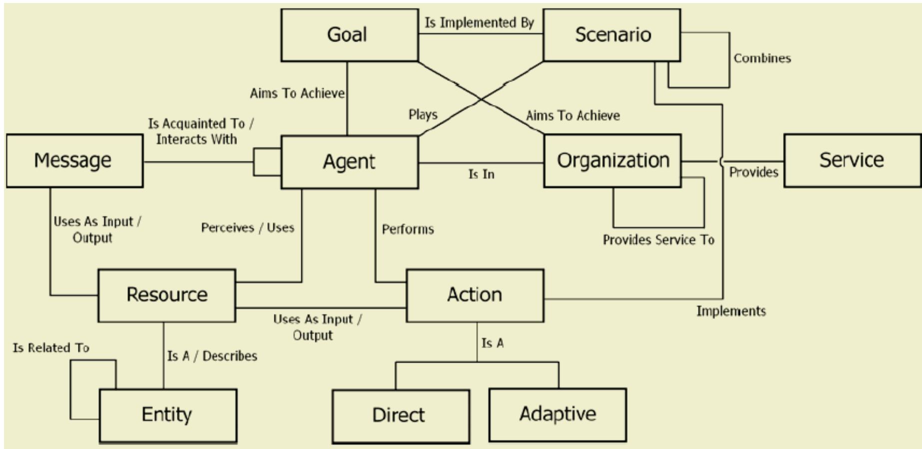


Fig. 1. The ANote conceptual meta-model

Goal. A Goal is a system objective, a functionality that must be achieved by one or more Agents. It associates an Agent with its Actions. Goals found in requirements can be of different categories. At the meta-level, such categories are organized into a specialization hierarchy, which means that goals can be refined into several alternative combinations of sub-goals.

Agent. An Agent is the main abstraction of the agent paradigm. An Agent can act and interact in the system in order to achieve a Goal. It has a limited perception of the system environment (i.e. Resources).

Scenario. A Scenario illustrates an Agent behavior (sequence of Actions) while it intends to accomplish a Goal in a particular context (system state), i.e. it shows a Goal realization. There can be usual contexts, which are contexts that show some usual execution of an Agent, and variant contexts, which are context that require some Agent adaptation (execution of other – possibly new – Actions).

Action. An Action is a computation that results in a change in the state of an Agent. Actions are the building blocks of Agents' action plans. There can be two types of Actions: DirectAction and AdaptiveAction. A DirectAction is an Action that an Agent usually performs while participating in a Scenario (context) to achieve a Goal. Nevertheless, if the context requires the Agent to adapt itself (this adaptation requires some reasoning functionality), the Agent may perform AdaptiveActions.

Message. A Message is conveyance of information from one Agent to another. It has a protocol (defined structure) and is mostly asynchronous.

Resource. Resource is used to represent non-autonomous Entities such as databases or external programs used by Agents. Resources describe the multi-agent system data (or ontology) level and they are useful to model the system environment; i.e., the set of data contents that Agents will manipulate while performing Actions.

Organization. An Organization is a group of (one or more) Agents working together in order to perform some useful function, i.e. deliver a Service. As there may be sev-

eral Organizations in a multi-agent system, Organizations are connected by provider/customer relationships. These relationships define how Agents in an Organization may depend on (interact with) Agents in another Organization.

3 ANote Views

A multi-agent system can be described, modeled and managed in terms of its structure and its dynamics. The system structure includes the agents, the environment, along with its resources, and also the organizational components (although the latter relates to a higher level of structure). The system dynamics are related to the descriptions of agents' behaviors. These two angles provide a model for agent-based system specification, which can be further detailed into views or perspectives. A view is a partial specification that provides a certain abstraction of a system aspect under consideration. It enables the software designer to concentrate on a single set of properties each time, as he only needs to write one view at a time. In addition, it enables him to consider only those features that are important for a particular context.

Each ANote view has a specific representation that supplies some properties, i.e. it is based on one of the ANote's meta-model concept. The combination of these properties provides an extended knowledge about a system specification [14].

3.1 Structural Views

The *Goal View* specifies the system goals. A goal defines a service or functionality that some user expects to get from the system. Goals lead to the incorporation of components that should support them and they may be used to assign the respective responsibilities of agents in the system, i.e. to provide the basis for defining which agents should best perform which actions.

Goals can be extracted from elaborations, process descriptions and the problem domain. The approach we use to elicit goals is the functional decomposition, very related to the approach popularized in [11, 38]. The basis for functional decomposition is to identify the functions that describe the system as a configuration of functional goals. Complex goals can be functionally decomposed into their constituent goals and flows, thereby providing a description as a hierarchical tree of goals.

Thus, this view provides an initial identification of a goal tree that outlines the functions performed by the entities that compose the system. In ANote, a goal is a node in the goal hierarchy tree and it is represented as a rectangle with rounded corners. Goals can participate in a specification relationship that is represented as an arrow that points from the specific node to the generic node (see Fig. 2).

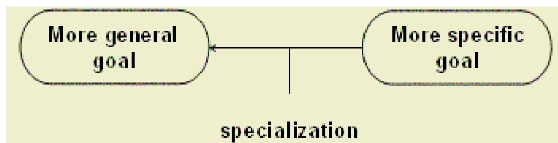


Fig. 2. Goal view notation

The *Agent View* specifies the agent types that exist in a multi-agent application solution and their relationships, thus defining the system structural base. The main

model element in this view is the agent class. An agent class is a description of a set of agents that share the same goals, functions, relationships, and semantics.

This view identifies the agents by a primary logical subdivision of the system's functional goals, defining the entities that will accomplish these goals. It is important to mention that an agent, in this view, is seen as a discrete modeling element, i.e. it provides no further details about the agent behavior in the system. The designer should be able to define the agent structure from the goal hierarchy, with the identification of agent classes as the roles that will be responsible to perform related functions in the system generic workflow. Thus, this view provides an elicitation of the agent types that may be refined later.

In ANote, an agent is represented as a rectangle with a tag A, for Agent. Two agents that interact in the multi-agent system participate in an association relationship. An association is represented as a line that links the agents that interact (see fig. 3).

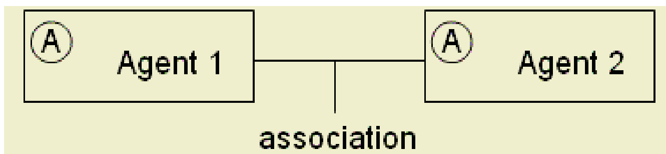


Fig. 3. Agent view notation

The *Ontology View* is the last static view of ANote. It identifies the non-agent components of the system, and it defines the world (i.e. environment) where the agents will execute to achieve their goals. This view provides a structural description of the agent environment resources.

Ontology can be used to specify the environment components with which the various agents operate, i.e., the information (or knowledge) manipulated by the agents and the relationships between them [3]. It includes the definitions of classes, relations, functions and other objects [18]. Therefore, a resource represents a particular instance of an object that has identity and attributes values and that can be manipulated by agents through its interface.

The approach to the environment resources specification is related to an object-oriented approach, such as [22]. It is based on the identification of the agent world specification as objects, with state and defined interactions with other objects. So, UML [28], together with its associated Object Constraint Language (OCL) [36], can be used as an alternative to represent the system environment. Some advantages of using a UML subset to model ontology can be seen in [4, 7].

Since ANote uses UML to model the system ontology, this view is described as a UML Class Diagram. An entity is represented as a class with the stereotype entity and a fact is represented as a class with the stereotype fact. Entities and facts can participate in the same relationships defined in the UML Class Diagram.

3.2 Dynamic Views

The *Scenario View* captures agent behavior in specific contexts. The main model element in this view is the scenario. A scenario is a description denoting similar parts

of possible agent behaviors limited to a context, i.e., to a purposeful state where actions and interactions take place among two or several agents. There are proposals [1, 21, 29] that interpret scenarios as containing information on how goals can be achieved. The goal-scenario combination has already been used to deploy goals [30], and this combination is specially fit to build a bridge between ANote's goal and scenario views. Scenarios also offer a concrete way to describe the circumstances in which a goal may fail, an agent may adapt or learn, and an agent may have an autonomous behavior. So, this view is also useful to show agency characteristics.

This view proposes to generate one or more scenarios from the previously done specifications. Each scenario is treated as one specific ordering of actions and events. Scenarios walk through each possible event sequence in a functional goal, applying heuristics which suggest possible exceptions that may occur at each step [34].

The scenario view helps the system developer to elaborate pathways through the goal in two phases. First, it generates each permissible normal course scenario for normal behavior, and then it identifies alternative paths for each normal sequence for emergent behavior (adaptive or exceptional context). Each pathway becomes a scenario. In ANote, scenarios are developed as goal schemas, resulting in a textual representation of how goals are achieved by agents. The goal schema has the following parts: main agent, prerequisites, usual action plan, interaction and variant action plan(s).

The *Planning View* specifies the execution states, or actions, an agent has to perform to compute an action plan. The main model element in this view is a graph, whose nodes are action states. An action state represents an action execution in a workflow. It means that the flow is waiting for the action to end in order to make a transition to the next set of possible action states. This view allows developers to model the basic workflows that are described in a scenario, and enable agents to deliver a service, i.e. accomplish a goal.

An action plan is modeled in a way that allows the agent to map it to its internal actions, to sequence the events for achieving a goal and to make decisions based on its current knowledge. The description of an agent's action plans comes from the courses of action (normal and alternatives) described in the scenario view. In ANote, action plans are represented as an action diagram very similar to a state chart [20]. It has action states and transitions. However, it introduces notation to represent agent adaptation with adaptive transitions to variant actions. Adaptive transitions, along with their tags, allow system designers to show when and under what circumstances an agent should change its behavior by executing a set of actions specified in the variant action plans of a scenario (see fig. 4).

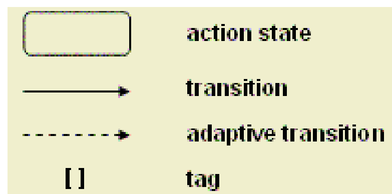


Fig. 4. Planning view notation

The *Interaction View* is used to represent the set of messages the agents exchange while computing an action plan. The main model element in this view is the message, which defines a particular communication between agents. A message is an asynchronous communication, i.e. the sender dispatches the message and immediately continues with the next step in the execution of its action plan. A message is an information flow between two agents, a sender and a receiver, and it may have a set of parameters. The message's name and parameters set define the message protocol.

This view shows the structural organization of agents that send and receive messages while executing an action plan. This description helps the designer to represent the messages and their protocols, in order to specify the communication infrastructure of the system. In ANote, interactions are represented as a conversation diagram, very likely to [2], in order to describe the discourse between agents (see fig. 5). Since messages in a multi-agent system are asynchronous, this diagram is important to: (i) show the current state of a conversation and, (ii) make a consistency that shows how the sends of an agent are matched by the receives of another agent.

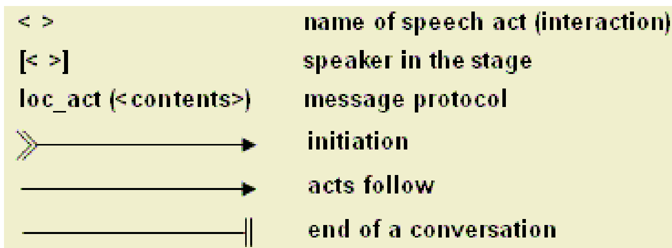


Fig. 5. Interaction view notation

3.3 Organizational View

The *Organizational View* defines the structure of a multi-agent system. This view specifies the system organizations and their relationships. The main model element in this view is the organization. An organization is an implementation unit that offers services (set of goals), accessed by an interface (set of message protocols). This view allows the system developer to think of a multi-agent system as a set of components or implementation physical units. The definition of organizations with precise interface specifications allows for their implementation to proceed independently from the implementation of the rest of the system.

There is no basic approach to split the agents into logical organizations – it is an arbitrary decision made by the software designer. So, the designer should decide which agents will be part of an organization and which properties (interface) will be visible or hidden.

In ANote, organizations are represented as boxes (see fig. 6). An organization box can show the set of agents that belong to it. Organizations can participate in a dependency relationship. A dependency shows that organizations are arranged in a client-server model. It expresses that an agent of an organization requires the service of an agent in another organization. A dependency is represented as a dashed arrow from the client to the server organization.

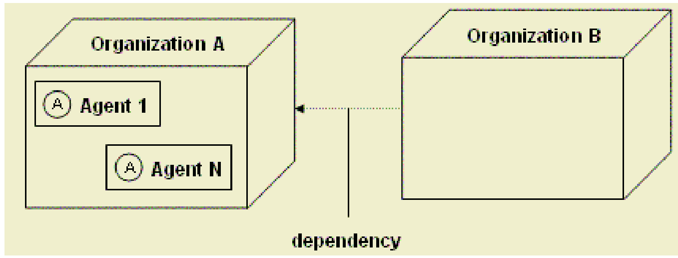


Fig. 6. Organization view notation

3.4 Consistency Between Views

In ANote, specification information is represented in a wide variety of representations, in different notations, and the information is possibly captured with different purposes. A variety of document views can be generated using ANote diagrams and, as the amount and diversity of information about the system grows, the need for supporting consistency and traceability among different levels of abstraction increases [19].

Thus, it is important to relate different diagrams, i.e. to give support for relating information across such representations. This includes support to retrieve information from within these diagrams, to navigate between them, and to handle changes made across them in a consistent manner. In fact, it is important to maintain the traceability of a concept that can appear in different views.

ANote has a set of rules [26], specified in OCL, to define the required consistency of a concept that may appear in diagrams from different views, and to serve as a logic basis to support consistency management. For instance, a goal must be accomplished by at least an agent. An agent accomplishes a goal by performing an action plan and, possibly, by interacting with other agents in a context. Thus, if a functional goal is defined, the designer will have to describe at least on scenario for it, with the constituents plans and interactions.

Another rule defines that if an agent appears in the interaction part of a scenario description, there must be an association, at the agent view diagram, between this particular agent and the lead agent. This set of rules is important to build management tools with embedded traceability and to ensure the robustness of the design. Since a multi-agent solution may be a large-scale system, management tools are important to support traceability between different representations, and to help visualize cross-representational concepts. The complete set of ANote's consistency rules is in [26].

4 A Case Study

In order to experiment with our modeling approach, we have experimented with several variations of agent-based specifications that are available in the literature. These include a marketplace [24] and a multi-agent system, the LearnAgents [31], for the Trading Agent Competition (TAC) [33].

The case study shown here is an insurance brokering system [25]. The example is a distributed, agent-based system that makes possible the electronic commerce of

insurance products. The system includes an agent representing each of the insurers, an agent representing the customer and a broker agent for brokering services. The customer (system user) provides the attributes of the needed insurance product to a customer agent. The customer agent sends this information to the broker agent. Then, the broker agent sends an announcement to all the insurer agents with the insurance product attributes.

When a broker agent sends an announcement, it starts a negotiation round with the insurer agents. At each negotiation round, insurer agents send proposals to the broker agent, which evaluates them. A negotiation round ends when a deadline is reached or a satisfactory proposal is received.

During the negotiation round, the broker agent may have received a set of satisfactory insurer proposals. So, the broker agent starts a new interaction with the customer agent to send those proposals. The customer agent displays each proposal to the user that can accept or reject it. If the user accepts a proposal, the customer agent starts an interaction with the insurer agent that made the proposal in order to buy it. The customer agent also updates a user profile that it keeps. If the broker agent did not receive any proposal, it sends a message to the customer agent indicating that no proposals are available. The customer agent can display this information to the user and finish (unsuccessfully) the execution, or it can relax some attributes of the needed product based on the profile of previous user interactions with the system and restart the negotiation process.

Observing a functional decomposition approach, the most generic goal of the system is to provide an insurance marketplace. In a first level of decomposition, the context goal can be subdivided into three goals: getting product info, negotiating product and delivering product.

The refinement of goals continues until the system developer states the functional goals. In this example, we consider the following functional goals: Retrieve product attributes; Update user profile; Inform product attributes; Announce product; Generate proposal; Evaluate proposal; Inform satisfactory proposals; Ask customer user satisfaction; Relax product attributes; Close deal. The hierarchy of goals produces the diagram of the goal view, as seen in fig. 7 (Negotiating product branch only, for brevity reasons).

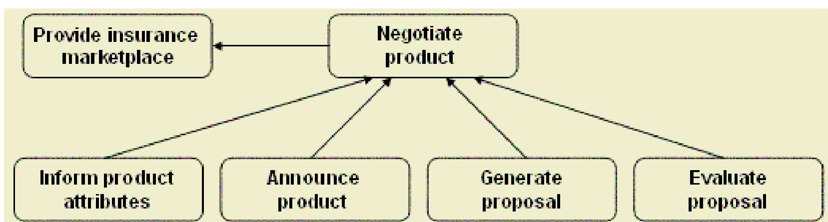


Fig. 7. Goal view diagram (partial)

The system agents are the customer agent, broker agent and insurer agent. Note that both customer and insurer agents interact with the broker agent. Thus, there must be an association relationship between customer and broker agents and between insurer and broker agents. The diagram for the agent view is seen in fig. 8. The system

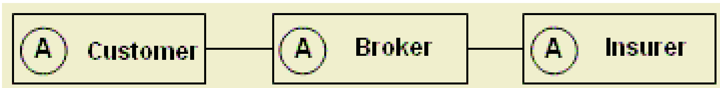


Fig. 8. Agent view diagram

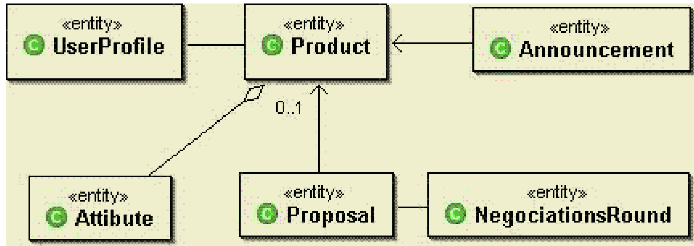


Fig. 9. Ontology view diagram

resources are related to insurance products, announcements, proposals, negotiation rounds and user profiles. The diagram for the ontology view is shown in fig. 9.

Now, the system designer must specify the scenarios. To specify a scenario, the system designer must relate a functional goal to an agent and must also describe the courses of action the agent may take to achieve the goal. For each functional goal of an agent there must be one, or more, scenario descriptions. The description of a course of action indicates when the agent needs to interact with another agent. So, the system designer must specify the interaction message (name and arguments). Actually, the diagrams for the scenario, planning and interaction views are closely related. Figures 10 to 12 show the diagram for a scenario and its corresponding planning and interaction diagrams.

SEND PROPOSAL	
MAIN AGENT	INSURER
PRECONDITION	Announcement received
USUAL ACTION PLAN	1. create list of Product Attributes 2. match with list of current available selling Products 2.1 for each matching selling Product 2.1.1 include selling Product in proposal list 3. for each Product in proposal list 3.1 generate Proposal 3.2 send Proposal
INTERACTION	BROKER
VARIANT ACTION PLAN	variant condition: no matching selling Product 1. pick selling Product closest related to announced Product by checking the Product Attribute set 2. generate Proposal with picked Product 3. send Proposal

Fig. 10. A scenario view diagram

Finally, the system developer may split the system into organizations. In this example, we decided to create just one organization. This is because the system is not too complex to require a division into more fine-grained organizations.

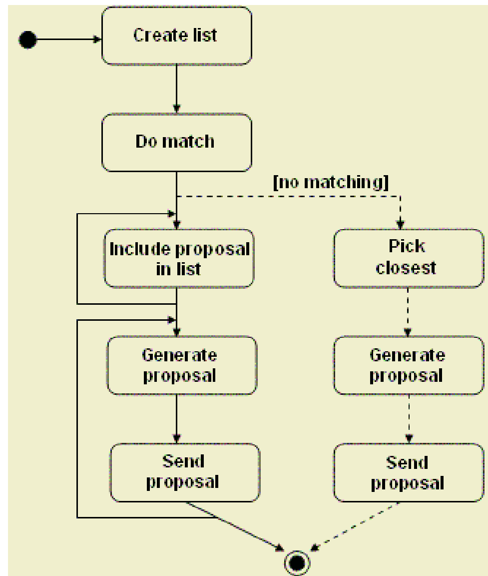


Fig. 11. A planning view diagram

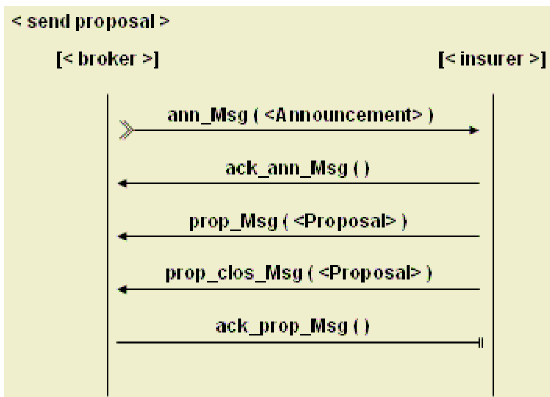


Fig. 12. An interaction view diagram

The diagrams presented above (again, the complete set of diagrams was not shown for brevity reasons) show part of the consistency rules there are between the ANote diagrams. For instance, the Send Proposal scenario shows the Broker agent class as participating in an interaction with the Insurer agent class. This information can also be seen in the association between the Broker and Insurer agent classes in the agent view diagram and in the interaction view diagram shown in Fig. 12.

5 Related Work

There is an extensive body of work being done in the field of agent-based systems modeling, most of it related to development methodologies. The Gaia [37] methodol-

ogy contains two analysis models and three design models. While the analysis models are based on well-defined concepts, agents are basically reactive since it does not model adaptation scenarios and plans. Thus, an agent always performs only its responsibilities with a rigid description of activities. Furthermore, Gaia specifies many different things in the Role Schema, which results in a scattered design of the system. For instance, it does not represent agency concepts and it mingles the specification of non-agent components with the specification of roles.

The Tropos [6] methodology focuses on the early requirements phase of software development. ANote provides a more detailed design process. In fact, it would certainly be possible to adapt Tropos early requirements phase for use in ANote, mostly in the goal elicitation and agent discovery phases. ANote goes further, by specifying the agent environment, the agents' action plans, interactions and organizations.

The MaSE methodology [9] is a conversation-centric methodology. In MaSE, agents coordinate their actions via conversations to accomplish individual and community goals. It is one of the few methodologies that appear to have significant tool support. However, MaSE is unsuitable for our purposes since it views agents "… merely as a convenient abstraction, which may or may not possess intelligence" [10, p232]. Thus, MaSE (intentionally) does not support the construction of plan-based agents that are able to provide a flexible mix of reactive and proactive behavior.

Some approaches are based on UML and extend or modify it, as in AUML [2], MESSAGE [5]. While UML, as an object-oriented modeling paradigm, is quite powerful, there are major conceptual differences between objects and agents, including differences in the degree of autonomy, flexibility and control [23]. AgentUML (AUML) [2], for example, defines extensions to UML Sequence Diagram with notations for agent concepts. However, just adding some new interaction modes does not turn an object into an agent. Besides, representing an agent as an object (i.e., as a set of attributes and methods) is not very useful because the object representation is too fine-grained and it is not at the same level of abstraction as the agent representation. For instance, a message in the object-oriented paradigm is a method invocation (i.e., the object always executes upon receiving). Nonetheless, in the agent-oriented paradigm, a message is asynchronous and should be dealt (agent planning) prior to execution (agent action).

MESSAGE [5] defines six agent models, and it is, probably, the most comprehensive method. Yet, it adopts UML and AUML, i.e. object-orientation, to detail the system design. In fact, using the object-oriented paradigm to model agent-based systems is not desirable since an agent and an object do not share the same characteristics. An object does not provide support to reactive and proactive behaviors. Furthermore, the object-oriented paradigm does not directly support an organizational structure, with asynchronous interactions and autonomous behavior.

It is true that ANote uses an UML diagram in the Ontology View. Nevertheless, in ANote, the UML Class Diagram is used to model the non-agent entities that inhabit the multi-agent system environment and that are used by agents. Note that this does not imply in a conceptual schizophrenia since object-orientation is never used to model agent concepts.

ANote offers a modeling language centered on the agent abstraction, based on the definition of a conceptual meta-model. This is very important to make developers think in terms of another paradigm. Also, ANote uses a goal-oriented approach to

specify an agent-based solution. The modeling language offers a logical division of a multi-agent system specification process in views, each one with its model. These models capture the agent system specifics, including the system functionalities (goal view), the agents that build the multi-agent solution (agent view), the environment modeling (ontology view), the way the resources in the environment will be used by agents (scenario and planning view), the agent planning (planning view), the multi-agent modules (organization view), and also agency properties, such as interaction (interaction view), autonomy (scenario and planning views), and the dynamic adaptation of the agents (scenario and planning views). In addition, ANote uses goal analysis techniques that have been shown to be very useful [8].

6 Conclusion

This paper presented ANote, a modeling language that offers a set of diagrams to model a multi-agent system in different views. A view includes the most important, or distinguishing aspects of something while suppressing or ignoring less important, immaterial, or diversionary details.

ANote's views are able to specify the system's structural (goal, agent and ontology) aspects, dynamic (scenario, planning and interaction) aspects and physical (organization) aspects. Besides, the views are interconnected in the sense that concepts from a view may appear on some other view diagram. This adds traceability to the concepts modeled in the diagrams.

ANote views facilitate a multi-agent system development in three different ways. First, they describe a multi-agent system using agent-based definitions, such as agents, plans, interactions, etc. Second, they illustrate abstractions by providing perspectives that encapsulate partial specifications, described in different notation sets. Finally, they increase modularity by decomposing a system into small, simple groupings, and also remove distinctions to emphasize commonalities.

The modeling language is used to specify multi-agent systems. The ANote was not developed with a particular architecture in mind since it intends to be general. However, it should guide the implementation into an agent architecture or framework. The works in [24, 31] show multi-agent systems that were modeled with ANote and implemented in two different architectures: one based on the CORBA Component Model [27] and another based on an Agent Framework [32].

The research on the ANote reported here is still in progress. Much remains to be done to further refine the proposed notation and validate its usefulness with other case studies. Experience in using the notation language to specify multi-agent systems is already contributing in this direction. Moreover, we believe that tool support will be a key aspect to help system developers to deploy multi-agent application solutions using ANote. There is already an ongoing work to provide a tool based on ANote, using the Eclipse [13] open platform. Since ANote has a conceptual meta-model, i.e. a common data model for all conceptual artifacts, and a set of consistency rules, this tool will also be able to help designers to develop consistent diagrams throughout the different views.

Acknowledgements

This work has been partially supported by the Software Engineering for Multi-Agent Systems (ESSMA) Project under grant 552068/2002-0 (CNPq, Brazil).

References

1. Antón, A.I., McCracken, W.M., Potts, C.: Goal Decomposition and Scenario Analysis in Business Process Reengineering. Proceedings of the Advanced Information Systems Engineering, CAiSE'94 (1994) 94-104.
2. Bauer, B., Muller, J., Odell, J.: Agent UML: A Formalism for Specifying Multiagent Software Systems. International Journal of Software Engineering and Knowledge Engineering 11(3) (2001) 207-230.
3. Bayardo, R.: InfoSleuth: Agent-Based Semantic Integration of Information in Open and Dynamic Environments. Proceedings of the ACM International Conference on the Management of Data, SIGMOD'97 (1997) 195-206.
4. Bergenti, F., Poggi, A.: Exploiting UML in the Design of Multi-Agent Systems. Proceedings of the 1st International Workshop Engineering Societies in the Agent World (2000) 106-113.
5. Caire, G., Coulier, W., Garijo, F., Gomez, J., Pavon, J., Leal, F., Chainho, P., Kearney, P., Stark, J., Evans, R., Massonet, P.: Agent Oriented Analysis Using Message/UML. In: Wooldridge, M.J., Weiss, G., Ciancarini, P. (eds.): Agent-Oriented Software Engineering II. Lecture Notes in Computer Science, Vol. 2222. Springer-Verlag, Berlin Heidelberg New York (2002) 119-135.
6. Castro, J., Kolp, M., Mylopoulos, J.: Towards Requirements-Driven Information Systems Engineering: the Tropos Project. Information Systems 27(6) (2002) 365-389.
7. Cranefield, S., Purvis, M.: UML as an Ontology Modeling Language. Proceedings of the IJCAI'99 Workshop on Intelligent Information Integration (1999) 46-53.
8. Dardenne, A., Lamsweerde, A., Fickas, S.: Goal-Directed Requirements Acquisition. Science of Computer Programming 20 (1993) 3-50.
9. DeLoach, S.A.: Multiagent Systems Engineering: a Methodology and Language for Designing Agent Systems. Proceedings of the 1st International Bi-Conference Workshop on Agent Oriented Information Systems, AOIS'99 (1999) 45-57.
10. DeLoach, S.A., Wood, M.F., Sparkman, C.H.: Multiagent Systems Engineering. International Journal of Software Engineering and Knowledge Engineering 11(3) (2001) 231–258.
11. De Marco, T.: Structured Analysis and Structured Specifications. Prentice Hall, New Jersey (1979).
12. Dieste, O., Juristo, N., Moreno, A., Pazos, J.: Conceptual Modeling in Software Engineering and Knowledge Engineering: Concepts, Techniques and Trends. In: Chang, S.K. (ed.): Handbook of Software Engineering and Knowledge Engineering Fundamentals. World Scientific Publishing Co., v. 1 (2001).
13. Eclipse.org: Eclipse, v 3.0 (2004).
14. Finkelstein, A.: Viewpoint Oriented Software Development: Methods and Viewpoints in Requirements Engineering. In: Bergstra, J.A., Feijs, L.M.G. (eds.): Algebraic Methods II: Theory, Tools, and Applications. Lecture Notes in Computer Science, Vol. 490. Springer-Verlag, Berlin Heidelberg New York (1991) 29-54.
15. Garcia, A.F., Lucena, C., Cowan, D.D.: Agents in Object-Oriented Software Engineering. Software: Practice and Experience 34(5) (2004) 489-521.
16. Garcia, A.F., Silva, V.T., Lucena, C.J.P., Milidiu, R.L.: An Aspect-Based Approach for Developing Multi-Agent Object-Oriented Systems. Proceedings of the 15th Brazilian Symposium on Software Engineering, SBES 2001 (2001) 177-192.

17. Genesereth, M., Ketchpel, S.: Software Agents. Communication of the ACM 37(7) (1994) 48-53.
18. Gruber, T.: A Translation Approach to Portable Ontology Specifications. Knowledge Acquisition 5(2) (1993) 199-220.
19. Grundy, J.C., Hosking, J.G., Mugridge, W.B.: Supporting Inconsistency Management for Multiple-View Software Development Environments. IEEE Transactions on Software Engineering 24(11) (1998) 960-981.
20. Harel, D.: StateCharts: A Visual Formalism for Complex Systems. Science of Computer Programming 8 (1987) 231-274.
21. Holbrook, C.: A Scenario-Based Methodology for Conducting Requirements Elicitation. ACM SIGSOFT, Software Engineering Notes 15(1) (1990) 95-104.
22. Jacobson, I., Booch, G., Rumbaugh, J.: The Unified Software Development Process. Addison-Wesley, Massachusetts (1999).
23. Kishore, R., Zhang, H., Ramesh, R.: A Helix-Spindle Model for Ontological Engineering. Communications of the ACM 47(2) (2004) 69-75.
24. Melo, F., Choren, R., Cerqueira, R., Lucena, C., Blois, M.: Deploying Agents with the CORBA Component Model. In: Emmerich, W., Wolf, A.L. (eds.): Component Deployment. Lecture Notes in Computer Science, Vol. 3083. Springer-Verlag, Berlin Heidelberg New York (2004) 234-247.
25. Nogueira, L., Oliveira, E.: A Multi-agent System for e-Insurance Brokering. Proceedings of the Workshop on Agent Technologies for e-Services, ATES 2002 (2002) 263-282.
26. Noya, R.C.: A Modeling Language for Multi-Agent Systems. PhD Thesis, PUC-Rio, Brazil (2002).
27. Object Management Group: CORBA Component Model, v 3.0 (2002).
28. Object Management Group: Unified Modeling Language, v 1.5 (2003).
29. Potts, C.: Fitness for Use: The System Quality That Matters Most. Proceedings of the International Workshop on Requirements Engineering (1997) 15-28.
30. Rolland, C., Souveyet, C., Achour, B.: Guiding Goal Modeling Using Scenarios. IEEE Transactions on Software Engineering 24(12) (1998) 1055-1071.
31. Sardinha, J.A.R.P., Choren, R., Lucena, C.J.P., Milidiú, R.L.: Engineering Machine Learning Techniques into Multi-Agent Systems. Submitted to the International Journal on Software Engineering and Knowledge Engineering (2004).
32. Sardinha, J.A.R.P., Ribeiro, P.C., Lucena, C.J.P., Milidiú, R.L.: An Object-Oriented Framework for Building Software Agents. Journal of Object Technology 2(1) (2003) 85-97.
33. SICS AB: Trading Agent Competition'04 (2004). See: http://www.sics.se/tac/news.php
34. Sutcliffe, A.: Supporting Scenario-Based Requirements Engineering. IEEE Transactions on Software Engineering 24(12) (1998) 1072-1088.
35. Tran, Q.N.N., Low, G., Williams, M.A.: A Preliminary Comparative Feature Analysis of Multi-agent Systems Development Methodologies. Proceedings of the 6th International Bi-Conference Workshop on Agent Oriented Information Systems, AOIS'04@CAiSE'04 (2004) 386-398.
36. Warmer, J., Kleppe, A.: The Object Constraint Language: Precise Modeling With UML. Addison-Wesley, Massachusetts, (1998).
37. Wooldridge, M., Jennings, N., Kinny, D. :The Gaia Methodology for Agent-Oriented Analysis and Design. Autonomous Agents and Multi-Agent Systems 3(3) (2000) 285-312.
38. Yourdon, E., Constantine, L.: Structured Design. Yourdon Press, New Jersey (1978).
39. Zambonelli, F.: Agent-Oriented Software Engineering for Internet Applications. In: Omicini, A., Zambonelli, F., Klusch, M., Tolksdorf, R. (eds.): Coordination of Internet Agents Models, Technologies, and Applications. Springer-Verlag, Berlin (2001) 326-346.

A Software Framework for Automated Negotiation*

Claudio Bartolini[1], Chris Preist[2], and Nicholas R. Jennings[3]

[1] HP Laboratories. Page Mill Rd., Palo Alto, CA 94304, USA
claudio.bartolini@hp.com
[2] HP Laboratories. Filton Road Store Gifford Bristol BS34 8QZ, UK
chris.priest@hp.com
[3] University of Southampton. Southampton SO17 1BJ, UK
nrj@ecs.soton.ac.uk

Abstract. If agents are to negotiate automatically with one another they must share a negotiation mechanism, specifying what possible actions each party can take at any given time, when negotiation terminates, and what is the structure of the resulting agreements. Current standardization activities such as FIPA [2] and WS-Agreement [3] represent this as a negotiation protocol specifying the flow of messages. However, they omit other aspects of the rules of negotiation (such as obliging a participant to improve on a previous offer), requiring these to be represented implicitly in an agent's design, potentially resulting incompatibility, maintenance and re-usability problems. In this chapter, we propose an alternative approach, allowing all of a mechanism to be formal and explicit. We present (i) a taxonomy of declarative rules which can be used to capture a wide variety of negotiation mechanisms in a principled and well-structured way; (ii) a simple interaction protocol, which is able to support any mechanism which can be captured using the declarative rules; (iii) a software framework for negotiation that allows agents to effectively participate in negotiations defined using our rule taxonomy and protocol and (iv) a language for expressing aspects of the negotiation based on OWL-Lite [4]. We provide examples of some of the mechanisms that the framework can support.

1 Introduction

Recently there has been much interest in the role of dynamic negotiation in electronic business transactions. For negotiation to take place between two or more parties, they need to agree on what economists refer to as a market mechanism or negotiation mechanism. This defines the rules of the "game" which the parties are engaged in and so determines the space of the possible actions that they can take. Within this game, each party adopts a strategy which determines exactly which actions they make (in response to actions by other parties or external events) in an effort to maximise their (individual or collective) gain. The mechanism must be public and shared by all parties, while an individual's strategy stays private, and is only revealed implicitly through the actions they take. For example, consider a simple market mechanism for an English auction. It is defined by the following rules: (i) the buyers can post bids at any time; (ii) a bid is only valid if it is higher than the currently highest bid; (iii) ter-

* This chapter is an updated and extended version of [1] C. Bartolini, C. Preist, N.R. Jennings *Architecting for Reuse: A Software Framework for Automated Negotiation*, in F. Giunchiglia, J. Odell, G. Weiß (Eds.) Agent-Oriented Software Engineering III, Springer-Verlag LNCS 2585/2003.

R. Choren et al. (Eds.): SELMAS 2004, LNCS 3390, pp. 213–235, 2005.
© Springer-Verlag Berlin Heidelberg 2005

mination occurs when no buyer has posted a bid in the last five minutes; (iv) after termination, the good is sold to the buyer with the current highest bid at the price bid.

The participants in the auction are constrained by these rules, but have a free choice of what action to take within them. A simple strategy for a buyer in such an auction is to set a maximum limit to the price they are willing to pay for the good, and to bid whenever the current highest bid is held by another buyer and is lower than their price limit.

In this chapter we consider mechanisms not strategies. In particular, we are concerned with the definition of interaction protocols underpinning a mechanism, rather than the emerging properties of the mechanism itself (for an example of the latter, compare [5]). The protocol determines the flow of messages between participants, specifying when an agent can send a message, and what messages it can send as valid responses to specified incoming messages. For example, a negotiation protocol for the English auction states that (among other things) that potential buyers send messages specifying their bids to the auctioneer, and receive an accept or reject message in response. When the auction terminates, all participants receive a message informing them of who the winner is, and the winning bid.

Various protocols are used for automated negotiation. They can be one-to-one (such as iterated bargaining [6]), one-to-many or many-to-many (such as auctions [7]). However, most state-of-the-art multi-agent systems are designed with a single negotiation protocol explicitly hard-coded in all agents (usually as finite state machines). This leads to an inflexible environment, only able to accept agents designed for it. An advance on this is provided by standardization activities such as FIPA [2] and WS-Agreement [3]. FIPA provides formal definitions of several standard negotiation protocols. The FIPA protocol for an English auction, described informally above, is shown in [8].

However, these negotiation protocols only formalise the interactions between the agents involved. They specify the permissible flow of messages, but omit information regarding other aspects of the rules of negotiation in a market mechanism.

For example, the FIPA English Auction protocol does not specify the criteria for a bid being acceptable (i.e. that it must be greater than the current highest bid) or the conditions under which the auction will terminate (i.e. that no bids have arrived in the last few minutes). Hence, because the multi-agent environment does not make these explicit, the designer of an agent using the protocol must be aware of these negotiation rules and design their agent taking them into account. As a result of this, with the exception of the interaction aspects, the negotiation mechanism is implicit in the design of the multi agent system [9].

All the considerations made above also apply WS-Agreement [3], a standard proposed by The Global Grid Forum (GGF). WS-Agreement includes the definition of a simple interaction protocol to support one-to-one negotiation, with the likely aim to support different mechanisms in the future through definition of multiple interaction protocols.

We propose an alternative to that currently adopted by FIPA and GGF. Our approach allows negotiation rules to be explicitly specified and categorised both at the design and at the implementation stage of agent oriented software development. We carry out an analysis of a generic negotiation process, which is able to capture common aspects of a wide variety of types of negotiation.

From there we derive: (i) a taxonomy of declarative rules which can be used to capture a wide variety of negotiation mechanisms in a principled and well-structured way and (ii) a simple interaction protocol, which is able to support any mechanism which can be captured using the declarative rules. This approach has the following advantages:

1. The generic negotiation process and rule taxonomy provide valuable conceptual tools for software engineers designing multi-agent systems which involve negotiation mechanisms. Their application will result in the mechanisms being represented in a more modular and explicit way than current approaches.
2. A set of rules together with an interaction protocol will fully specify a negotiation mechanism. Because of this, all information required for the design of agents using the negotiation mechanism is explicit and well-structured. This makes agent design and implementation easier, and reduces the risks of unintentional incorrect behaviour. This also opens the door for future research into creation and analysis of novel market mechanisms through exploration of new combinations of rules.
3. Because the rules specifying the negotiation mechanism are explicitly represented in the system, it is possible for an agent to reason over them to determine its behaviour and strategy. Ideally, an agent would be able to participate effectively in an arbitrary negotiation mechanism specified by any set of rules. Negotiation algorithms have been developed that are able to participate in several different negotiation mechanisms, and to adjust their behaviour depending on the details of the mechanism. For example, [10] present an agent algorithm able to simultaneously participate in multiple English, Dutch and Sealed Bid auctions, requiring details of bid increments, closing times and sealed bid winner announcement times to determine its exact behaviour. Using the negotiation framework that we present, an agent using such an algorithm could identify auctions of different types by checking the mechanism rules against templates, and could identify parameter values in the rules to determine the mechanism details.

To demonstrate the validity of our approach, in this chapter we also describe a software framework for automated negotiation that allows agents to effectively participate in negotiations defined using our rule taxonomy and protocol. The software framework can form a highly modular and reusable component in a multi-agent system. It advances the state of the art beyond the negotiation protocol approach because (i) it can be used to implement a wide variety of negotiation mechanisms simply by instantiating it with appropriate sets of rules. (ii) It is easy to maintain and update. If a software engineer determines that a particular negotiation must change its mechanism (see [11]), all they need do is adjust the rules appropriately. (iii) Agents involved in that negotiation can access the new rules, so at worst can identify that their current behaviour is inappropriate and issue a warning. A more advanced agent would be able to automatically modify their behaviour as necessary, provided the changes to the mechanism were not too great.

The remainder of this chapter is organized as follows: section 2 describes the generic negotiation framework, built upon the definition of an abstract negotiation process and a taxonomy for the rules of negotiation. Section 3 describes a prototype implementation of the negotiation framework. Section 4 presents a number of sample negotiation mechanisms that can be embodied by the framework. We discuss related work in section 5 and move to the conclusions in section 6.

2 The Generic Negotiation Framework

In this section, we present an abstraction of the negotiation process, developed from the analysis of many different negotiations, both automated and human. From this, we develop a general protocol for negotiation.

2.1 An Abstract Negotiation Process

The roles involved in the negotiation process are negotiation participant and negotiation host. In some market mechanisms participants address one another, whereas in others (e.g. auctions), participants send messages to a negotiation host that forwards them to other participants that have the right and interest in seeing them. Our abstraction is that participants always publish their proposals on a common multicast space, the negotiation locale, which is managed by the negotiation host. The negotiation locale can be considered as a form of blackboard, with access to write and visibility of information on it mediated by the negotiation host. Visibility rules are associated to proposals so that only the participants that have right to see them can see them. This allows us to see one-to-one and one-to-many negotiation as a particular case of many-to-many[1].

The agent playing the host role may also play a participant role (e.g. in one-to-one negotiation) or may be non-participatory (e.g. the auctioneer in an auction). In some cases, the role of negotiation host may alternate between different entities as the negotiation progresses.

The first action to be taken is for a participant to require admission to the negotiation. Much like in [13], admission consists of a simple conversation between the participant and the host where the participant requests admission to a particular negotiation and presents its credentials. Based on the credentials that the participant presents, the negotiation host decides whether to admit the participant to negotiation and informs the participant of the decision. If the participant is admitted, then we move onto the negotiation itself. The admission step is very important because it is when participants are informed of the rules of negotiation. To be able to negotiate with one another, parties must initially share a *negotiation template*. This specifies the different parameters of the negotiation (e.g. product type, price, supply date etc). Some parameters may be constrained (e.g. product type will almost always be constrained in some way), while others may be completely open (e.g. price). A negotiation locale has a negotiation template associated with it and this defines the object of negotiation within the locale.

As part of the admission process to the negotiation, participants must accept the negotiation template. The constraints expressed in the negotiation template remain static as the negotiation proceeds.

[1] This model always requires the participants to trust the negotiation host. Trust issues between participants and the negotiation host are addressed through the use of convertible undeniable signatures [12]. The imposition that proposals have to be signed with convertible undeniable signatures gives the protocol the following desirable properties. Even though proposals are invisible to the negotiation host, when an agreement is formed (i) participants cannot falsely claim ownership of the proposals and (ii) participants cannot repudiate the proposals that they have submitted, unless by refusing to collaborate in a revelation process.

The process of negotiation is the move from a negotiation template to an acceptable agreement. A single negotiation may involve many parties, resulting in several agreements between different parties and some parties who do not reach agreement. For example, a stock exchange can be viewed as a negotiation where many buyers and sellers meet to trade a given stock. Many agreements are formed between buyers and sellers, and some buyers.

During negotiation, the participants exchange *proposals* representing the agreements currently acceptable to them. Each proposal will contain constraints over some or all of the parameters expressed in the negotiation template. These proposals are sent to the negotiation host. However, before a proposal is accepted by the locale, it must be valid. To be valid, it must satisfy two criteria:

– It must be a valid restriction of the parameter space defined by the negotiation template. The constraints represent the values of parameters that are currently acceptable. Often, a constraint will consist of a single acceptable value.
– The proposal must be submitted according to the set of rules that govern the way the negotiation takes place. These rules specify (among other things) who can make proposals, when they can be made, and what proposals can be submitted in relation to previous submissions. For example, auctions often have a "bid improvement" rule that requires any new proposal to buy to be for a higher price than previous proposals. Such rules are specified and agreed at the admission stage.

An agreement is formed according to the agreement formation rules associated with the negotiation locale. When the proposals in the locale satisfy certain conditions, they are converted by these rules into agreements, and returned to the proposers. The end of a negotiation is determined by termination rules. For example, in an English auction the termination rule would state that the auction finishes when no participant has placed a bid for a certain time, and the agreement formation rule would state that an agreement is formed between the highest bidder and the seller, at the price the bidder has bid.

This abstract process can be specialised to many different negotiation styles. For example, in one-to-one bargaining, participants take turns in exchanging proposals in a previously agreed format. The rules in this case are simple. Any proposal can be made, as long as it is consistent with the negotiation template and made in turn. The negotiation terminates when the same proposal is returned unchanged (which we take as declaration of acceptance) or when one party leaves the negotiation locale. In the former case, an agreement identical to the last proposal is formed. In an English auction, the proposals specify the price of the good, every other parameter being fully instantiated in the negotiation template. Negotiation rules state that every new proposal (bid) will be valid only if it is an improvement over the current best proposal. Termination occurs at a deadline, and the agreement formed will contain the specification of the good as expressed in the negotiation template, at the price specified in the winning bid.

2.2 Taxonomy of Rules for Negotiation

So far we have been talking about negotiation rules in a very generic fashion. It is useful at this point to divide the negotiation rules into categories. By examining the

flexibility points of the abstract negotiation process described in the previous section, – for a more complete analysis see [14] – we identified the following categories of negotiation rules:

Rules for Admission of Participants
Admission Rules: Govern admission to negotiation.

Rules for Proposal Validity
Validity Rule: Enforces that any submitted proposal has to be compliant with the negotiation template.

Rules for Protocol Enforcement
Posting Rule: Determines when a participant may post a proposal.

Improvement Rule: Specifies, given a set of existing proposals, what new proposals may be posted.

Withdrawal Rule: Specifies if and when proposals can be withdrawn, and policies over the expiration time of proposals.

Rules for Updating Status and Informing Participants
Update Rules: Specifies how the parameters of the negotiation change on occurrence of certain events.

Visibility Rule: Specifies which participants can view a given proposal.

Display Rule: Specifies if and how the information updater notifies the participants that a proposal has been submitted or an agreement has been made - either by transmitting the proposal unchanged or by transmitting a summary of the situation.

Rules for Agreement Formation
Agreement Formation Rules: Determine, given a set of proposals of which at least two are compatible, which agreements should be formed.

Rules for Lifecycle of Negotiation
Termination Rule: Specifies when no more proposals may be posted (e.g. a given time, period of quiescence).

2.3 Definition of the Generic Negotiation Protocol

The three main phases of the generic negotiation protocol are: admission, proposal submission and agreement formation.

Admission Phase
We begin by describing the admission phase. The protocol requires the participant requesting admission to send an ACL.PROPOSE[2] message to the negotiation host. The payload of the message may contain credentials of the participant. The negotiation host replies either with an ACL.ACCEPT PROPOSAL or an ACL.REJECT PROPOSAL message, signifying admission (respectively rejection) of the participant to the negotiation. It has to be noted that this is a straightforward rendition of the

[2] We use FIPA ACL messages to describe the protocol. Other ACLs could equally be used.

FIPA propose interaction protocol, represented in figure 1 [8]. (Notice that in the FIPA protocol, our participant plays the role of the initiator, and the negotiation host plays the participant.)

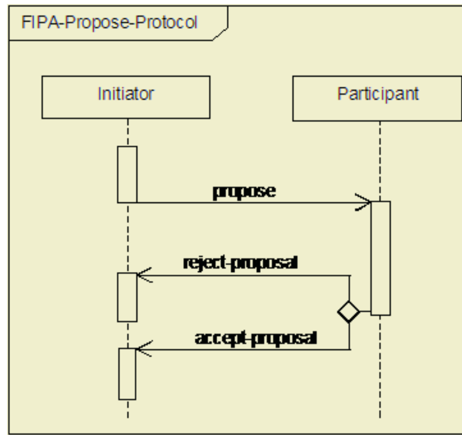


Fig. 1. The FIPA propose interaction protocol

Proposal Submission Phase

After admission, the participants submit proposals by posting them to the negotiation locale. Participants do so by sending an ACL.PROPOSE message to the negotiation host, whose payload contains the proposal. Proposal submission continues until termination is reached, as defined by the termination rules. Termination may occur after agreement formation (as in one-to-one bargaining), before agreement formation (as in a sealed bid auction) or may be independent (as in a continuous double auction). Each time a participant submits a proposal the negotiation host checks that it is syntactically well formed and it is a more constrained version of the negotiation template.

If the proposal is not valid, it is rejected. The submitter is notified with an ACL.REJECT PROPOSAL message. If the proposal passes this first stage of validation, the negotiation host checks that it satisfies the negotiation rules. These rules define the way in which the negotiation should take place and may include restrictions on when a proposal can be made (e.g. participants must take turns to submit) and semantic requirements on valid proposals (e.g. requirements that a proposal must improve on previous ones). If the proposal passes this second validation stage, the current set of proposals and associated data structures are updated accordingly and the submitter and other participants are notified. Who is notified, and the structure of the notification, is defined by the visibility rules and display rules. The submitter is notified through an ACL.ACCEPT PROPOSAL message. Once again, the protocol here described is compliant with the FIPA propose interaction protocol. Following the rules for updating negotiation status and informing participants, other participants may be notified through ACL.INFORM messages.

Agents submitting proposals may also withdraw proposals if the rules of negotiation allow them to. This is done through sending an ACL.CANCEL message where the communicative act that is being canceled is the previous instance of the proposal (all this is done according to the FIPA cancel meta-protocol [8].

Agreement Formation Phase

An agreement formation process can be triggered at any time during negotiation, according to the agreement formation rules. The negotiation host then looks at the current set of proposals to determine whether agreements can be made. Agreements can potentially occur whenever two or more negotiating parties make compatible proposals. If this is the case, agreement formation rules determine exactly which proposals are matched and the final instantiated agreement that will be used.

Agreement rules may state, for example, that the highest priced offer to buy should be matched with the lowest priced offer to sell and that the final agreement will take place at the average price. Often, tie breaking agreement rules will be defined that will be used if the main agreement rules can be applied in several ways. For example, earlier posted offers may take priority over later ones.

When the agreement formation rules have been applied to determine exactly which agreements are made, the negotiation host notifies the participants with ACL.INFORM messages.

Having defined the general protocol for negotiation (for a more complete specification and graphical representation, see [14]), we now show how it can be specialized in a variety of different ways. We do this firstly by presenting a taxonomy of negotiation rules and then (in the context of our prototype implementation) example rules for different negotiation mechanisms.

3 Implementation of the Software Framework

In our software framework, the negotiation host functionality is implemented by a responsible agent with a set of subsidiary agents. Each sub-agent is responsible for the enforcement of one of the categories of rules described in section 2.2: Gatekeeper (admission), Proposal Validator, Protocol Enforcer, Information Updater (updating status and informing participants), Negotiation Terminator (lifecycle of negotiation) and Agreement Maker. Each sub-agent interacts with other agents, both via direct messaging and by sharing data using a blackboard system. Any agent can join as a negotiation participant, provided it conforms to the generic negotiation protocol described in section 2.

The main task of the negotiation host agents is to evaluate negotiation rules and take actions as a consequence. To do so, they use the blackboard which contains information about the negotiation as a whole (e.g. valid proposals, participants, status of the negotiation). Each of the agents is initialized with the negotiation rules that it is responsible for enforcing. They execute rules either in response to a message or in response to changing data on the blackboard. Full details of the abstract architecture are given in [14].

We have implemented the negotiation framework using the Jade multi-agent platform [15]. Jade is compliant with the FIPA abstract architecture [2]. The main abstractions in Jade are agents and behaviours (section 3.1) Agents communicate using messages in the FIPA Agent Communication Language (ACL) [16]. Jade provides tools for inspecting these messages and also provides a library of interaction protocols and generic agent behaviours, which we have used as the basis of our implementation. The natural way of designing the negotiation host agents is as a rule engine. To do this we use the Java Expert System Shell (Jess).

Following [17], we associate a Jess rule engine with a Jade agent. We implement our negotiation rules in the Jess rule language. The agent's behavior monitors changes on the blackboard and incoming messages, and executes rules in response to these events.

Agents may write information about the negotiation on the blackboard (section 3.2). Proposals are also stored on the blackboard, provided they satisfy the negotiation template (section 3.3).

3.1 Agents and Behaviors

The Negotiation Host initializes the blackboard and creates the sub-ordinate agents. It acts as a first level contact for the negotiation participants. It receives proposals and forwards them to the Protocol Enforcer. Upon termination of the negotiation, it performs finalization tasks such as putting the agents to sleep. Each of the other agents has an associated Jess engine. When certain events occur (e.g. a new message or a change on the blackboard) they evaluate their rules and take the associated actions. This overall process is represented in Fig 2 (negotiate activity diagram).

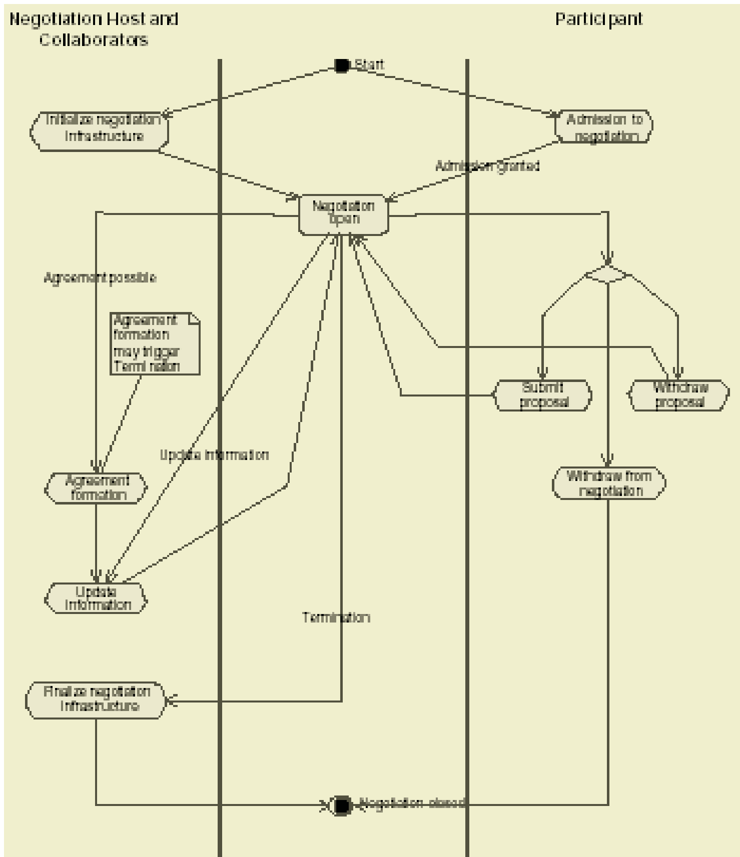


Fig. 2. Negotiate Activity Diagram

The Gatekeeper implements an agent-based version of a credentials-based access control system [18]. On receiving an ACL.REQUEST message from the Negotiation Host containing information on participant identity and credentials, it evaluates the admission rules to decide whether the participant should be admitted to negotiation. The Proposal Validator (Fig. 3) receives proposals (ACL.PROPOSE) from the Negotiation Host. It validates them against the negotiation template. If a proposal is valid, it forwards it to the Protocol Enforcer. Otherwise, it informs the submitter with an ACL.REJECT_PROPOSAL message. When the Protocol Enforcer receives a proposal from the Proposal Validator, it checks that the proposal satisfies the posting and improvement rules. It does this by invoking the Jess engine and accessing associated proposal data on the blackboard. If this succeeds, it declares the proposal valid and asserts it on the blackboard. The submitter is informed through an ACL.CONFIRM message with a proposal id. Otherwise it sends an ACL.REJECT_PROPOSAL message to the submitter. The Protocol Enforcer also processes withdrawal requests (ACL.REQUEST, where the payload is a proposal withdrawal referring to a valid proposal id), provided they satisfy the conditions of the withdrawal rules. The Negotiation Terminator regularly checks the termination rule to determine whether the negotiation should end. The termination rule is a Jess rule stating the conditions under which termination should occur (e.g. a time-out or following agreement formation). On negotiation termination, it notifies the Negotiation Host. At regular intervals or when a new proposal is posted on the locale, the Information Updater updates information on the blackboard appropriately. It may forward proposals to those participants eligible to see them (according to the visibility rules) and/or send a digest of the current state of the negotiation (according to the display rules).

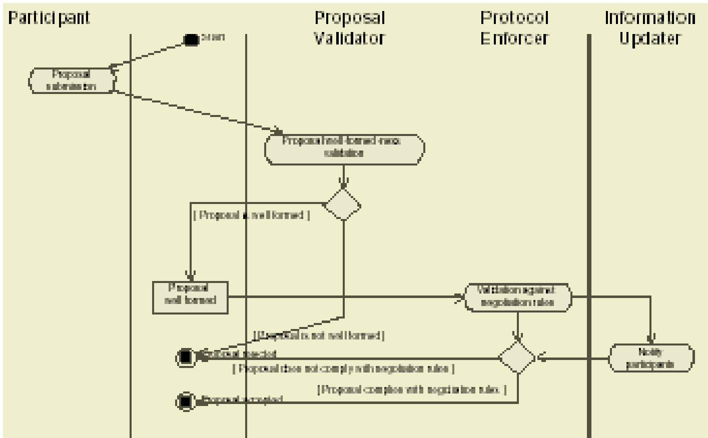


Fig. 3. Proposal Submission Activity Diagram

The Agreement Maker (Fig. 4) applies the agreement formation rules to determine which agreement can be made, given the valid proposals on the blackboard. It then notifies the interested participants that an agreement has been formed (ACL.INFORM). Its action can be triggered by an internal clock or by an event such as the arrival of a new proposal or the termination of the negotiation.

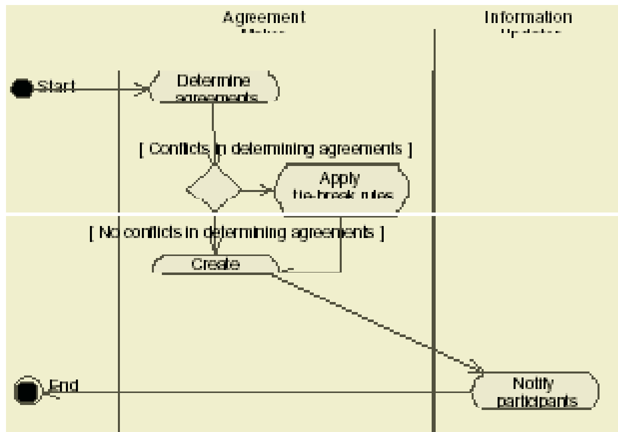


Fig. 4. Agreement Formation Activity Diagram

3.2 Assertions on the Blackboard

We now give details of the knowledge base used by the agents and then give details of the negotiation proposal language and negotiation rule language which make use of this. This knowledge base is stored in the negotiation locale and is accessible by the negotiation host and its sub-agents. All examples are given as Jess assertions and rules.

Facts About the Negotiation

The negotiation is assigned a unique ID at its start:

```
(negotiation (id Negotiation-Id))
```

Other parameters of the negotiation are asserted in the form

```
(negotiation
    (id Negotiation-Id)
    (negotiation-parameter Value))
```

For example, parameters associated with an English auction can be specified in the following way:

```
(negotiation
    (id auction-37)
    (seller-proposal Alice-37)
    (bid-increment 5)
    (termination-window 30min)
    (currently-highest-bid 0))
```

This states that auction-37 is selling a good described in proposal Alice-37 (See section 3.3), with an auction bid increment of 5. The first four fields will remain fixed, while the fifth will be updated regularly.

Facts About Participants

When a participant is admitted, the gatekeeper asserts relevant facts in the knowledge base. The participant is assigned an ID, and associated with a negotiation.

```
(participant
    (id Participant-Id)
    (negotiation-id Negotiation-Id))
```

Other parameters of the participants are asserted in the format:

```
(participant
    (id Participant-Id)
    (negotiation-id Negotiation-Id)
    (participant-attribute-name, Value))
```

For example, based on a participant's credentials, the gatekeeper may assign them a credit limit:

```
(participant
    (id Bob)
    (negotiation-id auction37)
    (creditLimit 10000))
```

Facts About Proposal Status

Facts are asserted which specify the current status of proposals on the blackboard. For example, when a proposal is first received, its submission time is asserted by the Gatekeeper as:

```
(submission-time 01/10/01:18:37
(proposal-id Proposal-Id))
```

When the proposal validator has checked a proposal, it asserts:

```
(valid-proposal
    (proposal-id Proposal-Id))
```

In a negotiation where new proposals can supersede old ones (such as an English auction), the Information Updater will assert facts specifying which proposals are currently active (and retract this if the proposal is superseded).

```
(active-proposal
    (proposal-id Proposal-Id))
```

3.3 Negotiation Proposals and Templates

The negotiation template and proposals are expressed as OWL-Lite descriptions [4]. We chose OWL-Lite because of its flexibility and expressiveness; the support that it offers for the creation and maintenance of ontologies and finally because it lends itself quite naturally to supporting the subsumption operation [19] that as we will see later is central to the functioning of our framework.

For a more in-depth discussion on why OWL-Lite and its precursor DAML+OIL [20] satisfy the requirements for a language for negotiation proposals and templates, see [19]. However, the choice of a description logic based language such as OWL-Lite is not to be intended as fundamental to the approach but is broadly indicative of what basic principles could be applied in designing the language.

For simplicity of exposition, here and in the following examples we will adopt a modified description logics notation [21] to express the proposals and the templates which is equivalent to the RDF OWL-Lite syntax [4]. XML Schema classes are not described but it should be clear by their names what they actually mean.

Before presenting the template and negotiation proposals, here are some descriptions of the concepts used. For brevity reasons, we will not exhaustively state all the description, but it should be quite intuitive to the reader what those concepts mean. For a more comprehensive description of the terms not defined here – such as Sale, Product and Participant descriptions for example – see [19].

The Car class is a subclass of Product and must have at most one Model and Make.

```
  Car ⊆ Product ∩
(=∃1 hasModel.Model) ∩
(=∃1 hasMake.Make)
  Model = {Punto, TT, S80}
  Make = {Ford, Audi, Volvo}
```

A negotiation host wishing to conduct auctions of cars could define the template as:

```
Template1 = Template ∩ Sale ∩
    ∀item.Car ∩
        ∀unitPrice.above2000 ∩
        ∀quantity.1 ∩
    ∀isComposedOf.(Delivery ∩ ∀date.before20041231)
```

A negotiation proposal must be a specialization of the negotiation template associated with the ongoing negotiation. According to the general protocol, negotiation participant agents can send proposals as ACL.PROPOSE messages containing a negotiation proposal specified as above. The Proposal Validator determines whether the proposal is valid with respect to (i.e. is subsumed by) the negotiation template by checking. An example of a proposal that is valid with respect to the template presented above is:

```
Proposal1 = Proposal ∩
    ∀seller.Alice ∩
    ∀item.(Car ∩ ∀hasMake.Fiat ∩ ∀hasModel.Punto) ∩
    ∀unitPrice.above3000 ∩ ∀quantity.1 ∩
∀isComposedOf.(Delivery ∩
∀date.between20041201and200412131))
```

This states that Alice – who is described as a participant in the participant ontology (see [19]) – wishes to sell a Fiat Punto for at least £3000 with delivery date after Dec, 1st 2004. The template requests that it also be specified that the delivery date be before the end of 2004.

When a negotiation terminates with an agreement acceptable to both parties, this agreement must specify the service that is going to be exchanged in an exact and non-ambiguous manner.

The main benefit of the choice of a description logic based language for expressing templates, proposals and agreements comes from the fact that the operations to be carried out over these descriptions by the subsidiary agents during the proposal validation and agreement formation phase can be reduced to the basic operations of checking for *satisfiability* and *subsumption* between descriptions that description logic reasoners can carry out effectively and efficiently [22].

Validation: The proposal validator, on receiving a proposal P, must initially check that it is valid. It is valid if it is a more constrained version of the negotiation template T for this negotiation. In description logic, this means that the negotiation host must check that T subsumes P. Formally, this can be specified as:

$$\mathtt{valid}_T\mathtt{(P)} \Leftrightarrow \mathtt{P} \subseteq \mathtt{T}$$

Agreement Formation: The agreement former come into action to identify all pairs of proposals which are compatible. Protocol specific rules are then used to determine exactly which of these pairs are used to form an agreement, and how exactly to generate the final agreement. A set of descriptions are compatible if their intersection is satisfiable:

$$\mathtt{compatible(D1,\ldots,Dn)} \Leftrightarrow \neg(\mathtt{D1} \cap \ldots \cap \mathtt{Dn} \subseteq \bot)$$

Hence, the first stage of agreement formation can be specified as follows:
Let Φ be the set of all valid proposals currently active on the negotiation locale.

$$\mathtt{potentialAgreements}(\Phi) = \{(P_i, P_j) \mid \mathtt{compatible}(P_i, P_j) \wedge i \neq j\}$$

When an agreement is formed, it can be verified a posteriori that the agreement subsumes the proposals that were used to form it and therefore the original negotiation template. Note that only two atomic operations are required to define the operations specified above:

- satisfiability $(\neg(X \subseteq \bot))$
- subsumption $(X \subseteq Y)$.

As noted above, a standard description logics reasoner is able to carry out both of these. Satisfiability lies at the core of such a reasoner, as all other reasoning or inference techniques are transformed into satisfiability checks. The subsumption operator is already defined by the OWL-Lite subClassOf, because our service descriptions are expressed as OWL-Lite classes (i.e. description logics concepts). A description logics reasoner can check whether two concepts subsume each other [22].

In the next section we give guidelines on how to write negotiation rules for various negotiation mechanisms.

3.4 Negotiation Rules

Subsidiary agents have standard rule templates, where the rule asserts information in their private fact base. The agent responds to this information, executing appropriate actions and sending messages according to the General Negotiation Protocol.

For example, the display rule in the Information Updater has the format:

```
(defrule display-rule ; declare the rule name
    (negotiation
        (...)) ; extract and process relevant parame-
ters
```

```
        from the DL description in the payload³
    => (assert
        (information-digest (...)))
    ; assert processed parameters to be published in
    the info digest
```

The visibility rules have a similar format, and act as filters on new proposals. They determine which participants can view which parameters of a new proposal. The information they assert is used by the Negotiation Host to mediate the view that different negotiation participants have on the blackboard.

```
(defrule visibility-rule
    (valid-proposal
        (...)) ; extract and process relevant parame-
ters
    (test (...)) ; test the required condition
    => (assert (visible-proposal (...)))
    ; if valid, assert that the proposal is visible
```

The termination rule in the Negotiation Terminator has the format:

```
(defrule termination-rule
    (...) ; extract and process relevant parameters
    (test (...)) ; test the termination condition
        => (assert (terminate <negotiation-id>)))
        ; if termination condition is met, assert nego-
tiation is terminated
```

Rules in the Protocol Enforcer (both posting and withdrawal) have a different format. Both when receiving protocols and withdrawal requests, the agent must check whether a series of conditions are all true to determine its action. Because of JESS´s cumbersome mechanism to support backward chaining, we implement these rules in the format:

```
(defrule <rule-name>
    (proposal (proposal-id ?Proposal-id)
        (...)) ; extract any other relevant parameters
    (test not(...)) ; REQUIRED CONDITION IN A NEGATED
                      FORM!!!
    => (assert (failed <rule-name> ?proposal-id)))
    ; if the condition is NOT met, assert the proposal
    is NOT valid
```

[3] In this example and in some of the following, we omit the adaptation code for extracting relevant parameters from the payload of the message that is sent from the participant to the message. As an example of how the parameters are processed, the proposal exemplified in section 3.3 would be asserted on the blackboard as:

```
(proposal
 (proposal-id Alice-37)
      ;ID is generated by the Negotiation Host
      (submitter Alice)
      (role Seller) ; Alice wishes to sell...
   (automobile
           (make FIAT) ;.. a FIAT Punto....
           (model Punto))
      (price ?P\&:(>= 3000 ?P))) ;... reservation price: 3000.
```

The Protocol Enforcer has a meta-rule which rejects the proposal if there are any such assertions in the database after the rules have executed, and accepts it otherwise. It executes appropriate actions and sends messages as defined in the General Negotiation Protocol.

4 Sample Mechanisms

In this section, we present a few examples of market mechanisms that the negotiation framework can support. For each of the mechanisms we give a flavor of the rules that need to be specified and the negotiation template and the negotiation proposals that participants may exchange.

4.1 Single Item English Auction

Assume a Negotiation Host has advertised an agreement template as per section 3.3, and has been contacted by Alice to sell her Fiat Punto via auction. The Host starts a new negotiation. It generates an associated agreement template, which is a specialized version of the one in 3.3, with the automobile slot instantiated with details of her Fiat Punto. The host asserts facts about the auction on the blackboard

The negotiation rules which apply to the seller state that they make a single proposal, and then remain silent. In the interests of space, we omit these. The proposal Alice makes is as specified in section 3.3. This confirms the details of the good she is selling, the expected delivery date, and specifies her reservation price of 3000. Facts about the auction are updated, and now appear as stated in the footnote[5] of section 3.4.

After this, buyers place bids in the form of proposals that satisfy the buyer proposal validation rules. These are applied by the Protocol Enforcer, and have the format described above (section 3.4). The conditions are:

[Posting rule] This tests that, if a buyer is posting a proposal, then the seller has already posted one.

```
(test (equal ?Role buyer)
    (exists (active-proposal (...) (role seller)))
```

[Improvement rule] The price field of the buyer's proposal must be a certain increment above the value of all previously posted buyer proposals. Hence the improvement rule contains the test:

```
(test (> ?Price (+ ?Currently-Highest-Price ?bid-
increment)))
```

[Withdrawal rule] Auctions do not allow bids to be withdrawn once submitted. Hence, the body of the withdrawal rule (in format specified earlier in this section - posting and withdrawal rules) contains `(test FALSE)` and so always fails when executed.

[Visibility rules] The seller's initial proposal is visible to all the buyers.
However, the field in which the seller constrains the price to be above their reservation price cannot be viewed:

```
(defrule visibility-rule
    (active-proposal(proposal-id ?PID)(role seller))
    (test(TRUE))
        => (assert
            (visible-proposal
                (proposal-id
                     (value ?PID)
                     (visibility all))
                (price
                     (value ?Price)
                     (visibility none))
                (...)))
```

A similarly structured rule states that all active buyer proposals are visible to all participants. Optionally, the identity of a bidder can be maintained private.

[Display rule] The currently highest bid price is notified to all participants.

```
(defrule display-rule
    (negotiation
        (...)
        (currently-highest-bid ?CHB))
    => (assert
        (information-digest
            (currently-highest-bid ?CHB)))
```

[Termination rule] Termination occurs if the auction is inactive for longer than the termination window specified in the negotiation fact base. Hence the rule, in the format specified in the beginning of this section, contains the test:

```
(test (> ?Current-Time (+?Active-Proposal-Time ?Termi-
nation-Window))
```

Together with the information asserted in section 3, this results in Alice's auction terminating if it is inactive for 30 minutes.

[Agreement formation rules] When negotiation terminates, an agreement is formed between the currently active buyer and the seller. The agreement states that the item specified in the template is sold to the buyer at the price specified in the currently active proposal.

```
(defrule agreement-formation-rule
    (active-proposal
        (proposal-id ?B-PID) (submitter ?BUYER)
        (role buyer) (price ?PRICE))
    (active-proposal
        (proposal-id ?S-PID) (submitter ?SELLER)
        (role seller) (price ?RES-PRICE))
    (test
        (> PRICE RES-PRICE))
            => (assert
                (agreement
                    (buyer ?BUYER) (seller ?SELLER)
                    (price ?PRICE))))
```

4.2 The Continuous Double Auction

A many-to-many Continuous Double Auction can be implemented in our framework by straightforward modification of the rules above. For example, the improvement rule requires new bids/offers to be higher/lower than the currently active bid/offer.

We have one rule which matches with seller proposals, with test:

```
(test (> ?Price ?Currently-Lowest-Offer))
```

and a simlar rule for buyer proposals with test:

```
(test (> ?Price ?Currently-Highest-Bid))
```

The posting rule is modified to allow both buyer and seller proposals at any time. In addition to the highest bid, the information digest also contains the lowest offer. Termination occurs at a fixed time, so the test becomes:

```
(test (> ?Current-Time ?End-Time))
```

The only substantial change is in the agreement formation rule. Agreement is formed whenever there is a bid greater than an offer.

Highest bids are matched with lowest offers, with the agreement at the midpoint.

```
(defrule agreement-formation-rule
    (active-proposal
        (proposal-id ?Seller-PID)
        (price ?Seller-price))
    (active-proposal
        (proposal-id ?Buyer-PID)
        (price ?Buyer-price))
    (currently-highest-bid ?Buyer-Price)
    (currently-highest-ask ?Seller-Price)
        => (assert
           (agreement
               (proposals
                   (?Seller-PID ?Buyer-PID))
               (price (= (/ 2 (+ (?BP ?SP))…)))
```

After an agreement is made, the Information Updater will declare the next highest/lowest bid/offer to be active. This may result in more agreements being formed immediately.

4.3 Simple Shop Front

The framework can also model one-to-one negotiation such as a simple shop front. In this example the shop is a car dealership. The actors involved in the simple car dealership scenario are the car dealer and one or more buyers. A prospective buyer plays the participant role, whereas the shopkeeper plays both the participant and the negotiation host roles at the same time. The car dealership is modeled following the negotiation locale abstraction.

Before negotiation begins, the shopkeeper decides the admission policy, negotiation template, and negotiation and agreement formation rules.

Once again the template is identical to the one in the example given in section 3.3, expressing the cars that the dealer is willing to sell, minimum price and earliest delivery date.

The car dealer adopts standard ‘shop front take it or leave it’ negotiation rules. These state that[4]:

[Posting rule] A buyer may post a proposal at any time, irrespective of posted proposals by other buyers. A seller may post proposals at any time.

[Termination rule] Termination occurs when there are no seller proposals posted in the shop front

[Withdrawal rule] A seller may withdraw proposals at any time so long as they have not been matched yet with buyer’s proposals. Proposals from the buyers are committing, so buyers cannot ever withdraw proposals.

The car dealer adopts standard shop front agreement formation rule:

[Agreement formation rule] Agreements are formed whenever a buyer posts a proposal identical to the seller’s proposal.

After rules have been specified, negotiation can begin. The car dealer in its seller role (Alice) submits proposals for all goods it sells. The seller’s proposals take a similar form as the example given in section 3.3.

If it expects high demand, it can place several identical proposals on the table for the same good. If all proposals for a given good are accepted, and the car dealer still has more in stock, it resubmits identical proposals. A buyer submits a proposal, an identical copy of the car dealer’s proposal, when it wishes to purchase a given good. Agreement formation occurs as the car dealer– in the referee role – identifies valid buyer proposals and sends agreements to the buyers.

4.4 Multi-party Contracts

The examples given so far addressed the formation of two-party contracts, whatever the number of participants. The negotiation framework though extends quite naturally to the case of agreements among multiple parties playing different roles, noting a couple of observations.

To begin with, admission can be conditioned to being able to bid for one or more roles. Participants submit proposals specifying the role they want to play, selected from the role (or roles) for which they have been admitted. The proposals may also constrain who should (or should not) play the other roles.

Secondly, visibility rules enforce that participants that have been admitted to play a certain role have a restricted view over other participant’s proposals. Each participant will only be able to see the part of the other proposals that are directly relevant to the role they want to fulfill. This enables entities to propose modifications to relevant parts of the contract without having access to other non-relevant parts. When all parties have agreed, each will have proposed a partly-instantiated contract that is consistent with all the others and hence the negotiation host will be able to produce the final contract according to the agreement formation rules.

As an example, imagine that a multi-party agreement is sought between a building contractor and other participants to fulfill the role of a carpenter, builder, and electri-

[4] For this example we do not present the rules in Jess language for reasons of space. However, they are similar enough to the ones in the two previous examples that the attentive reader will not have problems deriving them.

cian. Participants are admitted to the negotiation bidding to undertake the roles that they specialize for. The agreement template will include general information accessible to all parties, such as general recital information, boiler plate terms etc. Other parts of the agreement template might be restricted to fewer roles. The rest of the negotiation process is carried out exactly as described in section 2.1. The only difference is that the resulting agreement will concern more than two roles which therefore will be assigned to more than two participants.

5 Related Work

Research on agent negotiation protocols has primarily focused on the specification of specific protocols, often using conversations [23] specified as finite state machines. For example, Parsons et. al. define a flexible protocol for one-to-one bargaining using this approach [6]. The FIPA agent standardization effort has defined various interaction protocols, including English and Dutch auctions, as interchanges of messages in FIPA ACL [8]. These are effectively a set of one-to-one conversations which must be coordinated. Pitt et. al.[24] define a semantic framework around FIPA ACL to allow the easier specification of multi-party interactions by adding structured conversation identifiers and a richer representation of protocol states. WS-Agreement [3] defines a simple interaction protocol aimed at supporting one-to-one negotiation. Our approach differs from these in that rather than defining a library of protocols, we define a general protocol that can be parameterized with rules.

Research in negotiation in the semantic web domain spun from the concern of demonstrating that semantic web languages can provide useful semantic support to the processes of matchmaking and negotiation [19] therefore only marginally touching on the problem of defining interaction protocols.

Naftaly Minsky's Law Governed Interactions (LGI) [25] is a paradigm for agent co-ordination that can presents similarities to our approach. However, the scope of LGI is much wider than just negotiation and applies to a much wider variety of coordination mechanisms. It's true that LGI has been applied to peer-to-peer auctions [26], but the focus of that work was mainly on the peer-to-peer aspect, aiming to dispense with a centralized service for auctions. In contrast, our framework is especially designed for providing a protocol that can embody multiple negotiation mechanisms. In this chapter, we describe a reference implementation for the framework based on Jess, but one could envisage populating the taxomomy of negotiation rules that we propose through LGI laws. Similar considerations apply to comparing the framework here described with the work of Artikis et al. [27].

Esteva et.al. [13] have defined a formal approach to specifying electronic institutions in which agents interact. This goes beyond other work on protocols in the additional abstractions it provides. It associates different protocols to scenes, and provides means for specifying transition conditions from one scene to another together with normative rules associated with transition. Our work is complementary to this, in that our focus is primarily on a single scene (negotiation) and providing flexibility within it.

Reeves et. al. [28] have also built on this to configure a general auction server with auction rules and contract templates. Their architecture is server-based, rather than agent-based, and participant agents must still be hard-coded with specific protocols.

Our general negotiation protocol allows us to handle richer negotiation mechanisms than they support. Other architectures for negotiating agents have been proposed [29] that present a neater separation of concerns between the definition of the protocols according to the principles described in this chapter and the construction of the negotiating agents.

Wurman et. al. [30] carried out a thorough analysis of the auction design space, classifying auction mechanisms according to different parameters. This work, focusing primarily on auction rules, provided valuable input to our analysis.

Mechanism design has recently had a surge in popularity [15, 5] as a foundation for building multi-agent software systems. We envisage that the generic negotiation framework described in this chapter could provide a useful platform for experimenting with it, given the flexibility that it provides for the declarative definition of interaction protocols underpinning different the negotiation mechanisms.

6 Conclusions

In this chapter, we have discussed the shortcomings of the representation of negotiation mechanisms in standardization activities such as FIPA [2] and the Global Grid Forum's (GGF) WS-Agreement [3]. Specifically, we have shown that the protocol approach adopted by them and many others results in only part of a mechanism being explicitly formalised and standardised, which can result in significant drawbacks from a software engineering perspective. Alternatively, we propose a modular approach to negotiation mechanisms: a generalized interaction protocol which can be specialised with declarative rules. We provide a taxonomy of such rules and a software framework that implements this approach and give examples of rules for various negotiation mechanisms. The aim of our framework is to go beyond what is currently offered by the existing standards, to provide a flexible approach to defining negotiation protocols enforcing the rules of the negotiation without having to adopt a fully-fledged coordination mechanism à la LGI [25]. We believe that our framework covers a wide variety of negotiation mechanisms – of which we give a flavor in section 4 - and gives a mechanism designer the possibility of easily creating new combination of negotiation rules.

References

1. Bartolini, C., Preist, C., Jennings N.R.: Architecting for Reuse: A Software Framework for Automated Negotiation, in Giunchiglia, F., Odell, J., Weiss G. (Eds.): Agent-Oriented Software Engineering III (2002), Springer-Verlag LNCS 2585/2003.
2. Foundation for Physical Agents. Fipa abstract architecture specification, 2000. Available at www.fipa.org.
3. Andrieux A.et. Al.: Web-Services Agreement Specification (WS-Agreement) Global Grid Forum Recommendation. Available at www.ggf.org
4. Dean, M., Schreiber G.: OWL Web Ontology Language Reference W3C Recommendation. Available at www.w3c.org
5. Dash, R. K, Jennings, N. R., Parks, D. C.: Computational Mechanism Design: A Call to Arms. IEEE Intelligent Systems (2003), vol. 18 (6), 40-47.
6. Parsons, S., Sierra, C., Jennings, N.R.: Agents that reason and negotiate by arguing. Journal of Logic and Computation (1998), 8(3), 261–292.

7. Wurman, P.R., Wellman, M. P., Walsh, W.E.: The Michigan Internet AuctionBot: A configurable auction server for human and software agents. In Sycara, K. P., Wooldridge M. (eds), Proceedings of the 2nd International Conference on Autonomous Agents (Agents'98) (1998), 301–308, New York, 9–13, 1998. ACM Press.
8. Foundation for Physical Agents. FIPA Interaction Protocol Library Specification, 2000. Available at www.fipa.org
9. Jennings, N. R., Norman, T. J., Faratin, P.: ADEPT: An agent-based approach to business process management. ACM SIGMOD Record (1998), 27(4), 32–39.
10. Byde, A., Preist, C., Jennings, N.R.: Decision procedures for multiple auctions. In Proceedings of the 1st Joint International Conference on Autonomous Agents and Multi-Agent Systems (2002), 613-620.
11. Sandholm, T.: Automated Mechanism Design: A New Application Area for Search Algorithms. In proceedings International Conference on Principles and Practice of Constraint Programming (CP-03), 2003.
12. Boyar. J., Chaum, D., Damgard, I. Pedersen, T.: Convertible Undeniable Signatures; Crypto '90, LNCS 537, Springer-Verlag, Berlin (1991), 189-205.
13. Esteva, M., Rodriguez, J. A., Sierra, C., Garcia, P., Arcos, J. L.,: On the formal specifications of electronic institutions, In Dignum F. Sierra, C. (eds.) Agent-mediated Electronic commerce (The European AgentLink Perspective), Springer LNAI. (2000)
14. Bartolini, C. Preist, C., Jennings N.R.: A Generic Software Framework for Automated Negotiation, HP Laboratories Technical Report HPL-2002-2, (2002)
15. Bellifemmine, F., Poggi, A., and Rimassa, G. Jade - A FIPA compliant Agent Framework. In Proc. 4th International Conference on Practical Applications of Intelligent Agents and Multi-Agent Systems (1999)
16. Foundation for Physical Agents. FIPA ACL Message Structure Specification, 2000. Available at www.fipa.org
17. Hoffmann, O., Stumptner, M., Chalabi, T.: A perspective based approach to design. In Workshop on Planning, Scheduling and Configuration, KI2001 (2001)
18. Bartolini, C. and Casassa-Mont M, Digital Credentials and Authorization to Enhance Trust in Negotiation within E-services Marketplaces. In Proc. 7th HP Openview University Association Plenary Workshop (2000).
19. Trastour, D., Bartolini, C., Preist C.: Semantic Web Support for the Business-to-business E-Commerce Lifecycle In Computer Networks: The International Journal of Computer and Telecommunications Networking, Vol. 42, Issue 5 (2003) Special Issue on The Semantic Web: an Evolution for a Revolution - North Holland / Elsevier
20. van Harmelen, F. and Horrocks, I. Reference Description of the DAML+OIL Markup Language. Available from www.daml.org, (2000)
21. Baaders, F., Calvanese, D., McGuinness, D., Nardi, D., Patel-Schneider, P.: The Description Logic Handbook. Cambridge University Press (2002)
22. Horrocks I, Patel-Schneider, P.F.: Comparing subsumption optimizations. In Franconi, E. De Giacomo, G. MacGregor, R.M., Nutt, W. Welty, C.A., Sebastiani, F. (eds), Collected Papers from the International Description Logics Workshop (DL'98), pages 90–94. CEUR, (1998).
23. Barbuceanu, M. and Fox, M.S. COOL: A language for describing coordination in multi-agent systems. In Proc. First International Conference on Multi-Agent Systems, MIT Press, (1995), 17-24.
24. Pitt, J., Guerin, F. and Stergiou, C.: Protocols and Intentional Specifications of Multi-Party Agent Conversations for Brokerage and Auctions. In Proc. Fourth International Conference on Autonomous Agents, ACM Press (2000), 269-276
25. Minsky, N. and Ungureanu, V. Law-Governed Interaction: A Coordination & Control Mechanism for Heterogeneous Distributed Systems. in ACM Transactions on Software Engineering and Methodology (TOSEM) Vol 9.3, 273-305 (2000)

26. Fontoura, M., Ionesu, M. and Minsky N. Law-Governed Peer-to-Peer Auctions In Proc. of the eleventh international world wide web conference (WWW2002) Honolulu, Hawaii, May (2002)
27. Artikis A., Sergot M. and Pitt J. Specifying Electronic Societies with the Causal Calculator. In the Proceedings of the Agent-Oriented Software Engineering III (AOSE) workshop, LNCS 2585, Springer, (2003)
28. Reeves, D., Wellman, M. and Grosof, B. Automated Negotiation from Declarative Contract Descriptions. In Proc. Fifth International Conference on Autonomous Agents, (2001)
29. Ashri, R., Rahwan, I. and Luck, M.: Architectures for Negotiating Agents, in Mueller, M.V. Pechoucek, J.(eds). Multi-Agent Systems and Applications III, pages pp. 136-146. Springer, (2003)
30. Wurman, P, Wellman, M. and Walsh W.: A Parameterization of the Auction Design Space, in Games and Economic Behavior, 35 Vol. 1/2 (2001), 271-303

Efficient Agent Communication in Multi-agent Systems

Myeong-Wuk Jang, Amr Ahmed, and Gul Agha

Department of Computer Science
University of Illinois at Urbana-Champaign,
Urbana IL 61801, USA
{mjang,amrmomen,agha}@uiuc.edu

Abstract. In open multi-agent systems, agents are mobile and may leave or enter the system. This dynamicity results in two closely related agent communication problems, namely, efficient message passing and service agent discovery. This paper describes how these problems are addressed in the *Actor Architecture* (*AA*). Agents in AA obey the operational semantics of actors, and the architecture is designed to support large-scale open multi-agent systems. Efficient message passing is facilitated by the use of dynamic names: a part of the mobile agent name is a function of the platform that currently hosts the agent. To facilitate service agent discovery, middle agents support application agent-oriented matchmaking and brokering services. The middle agents may accept search objects to enable customization of searches; this reduces communication overhead in discovering service agents when the matching criteria are complex. The use of mobile search objects creates a security threat, as codes developed by different groups may be moved to the same middle agent. This threat is mitigated by restricting which operations a migrated object is allowed to perform. We describes an empirical evaluation of these ideas using a large scale multi-agent UAV (Unmanned Aerial Vehicle) simulation that was developed using AA.

1 Introduction

In open agent systems, new agents may be created and agents may move from one computer node to another. With the growth of computational power and network bandwidth, large-scale open agent systems are a promising technology to support coordinated computing. For example, agent mobility can facilitate efficient collaboration with agents on a particular node. A number of multi-agent systems, such as EMAF [3], JADE [4], InfoSleuth [16], and OAA [8], support open agent systems. However, before the vision of scalable open agent systems can be realized, two closely related problems must be addressed:

- *Message Passing Problem*: In mobile agent systems, efficiently sending messages to an agent is not simple because they move continuously from one agent platform to another. For example, the *original agent platform* on which an agent is created should manage the location information about the agent.

R. Choren et al. (Eds.): SELMAS 2004, LNCS 3390, pp. 236–253, 2005.
© Springer-Verlag Berlin Heidelberg 2005

However, doing so not only increases the message passing overhead, but it slows down the agent's migration: before migrating, the agent's current host platform must inform the the original platform of the move and may wait for an acknowledgement before enabling the agent.

– *Service Agent Discovery Problem*: In an open agent system, the mail addresses or names of all agents are not globally known. Thus an agent may not have the addresses of other agents with whom it needs to communicate. To address this difficulty, middle agent services, such as *brokering* and *matchmaking* services [25], need to be supported. However, current middle agent systems suffer from two problems: *lack of expressiveness*–not all search queries can be expressed using the middle agent supported primitives; and *incomplete information*–a middle agent does not possess the necessary information to answer a user query.

We address the message passing problem for mobile agents in part by providing a richer name structure: the names of agents include information about their current location. When an agent moves, the location information in its name is updated by the platform that currently hosts the agent. When the new name is transmitted, the location information is used by other platforms to find the current location of that agent if it is the receiver of a message. We address the service agent discovery problem in large-scale open agent systems by allowing client agents to send search objects to be executed in the middle agent address space. By allowing agents to send their own search algorithms, this mitigates both the lack of expressiveness and incomplete information.

We have implemented these ideas in a Java-based agent system called the *Actor Architecture* (or *AA*). AA supports the *actor semantics* for agents: each agent is an autonomous object with a unique name (address), message passing between agents is asynchronous, new agents may be dynamically created, and agent names may be communicated [1]. AA has been designed with a modular, extensible, and application-independent structure. While AA is being used to develop tools to facilitate large-scale simulations, it may also be used for other large-scale open agent applications. The primary features of AA are: a light-weight implementation of agents, reduced communication overhead between agents, and improved expressiveness of middle agents.

This paper is organized as follows. Section 2 introduces the overall structure and functions of AA as well as the agent life cycle model in AA. Section 3 explains our solutions to reduce the message passing overhead for mobile agents in AA, while Section 4 shows how the search object of AA extends the basic middle agent model. Section 5 descries the experimental setting and presents an evaluation of our approaches. Related work is explained in Section 6, and finally, Section 7 concludes this paper with future research directions.

2 The Actor Architecture

AA provides a light-weight implementation of agents as active objects or actors [1]. Agents in AA are implemented as threads instead of processes. They

use object-based messages instead of string-based messages, and hence, they do not need to parse or interpret a given string message, and may use the type information of each field in a delivered message. The actor model provides the infrastructure for a variety of agent systems; actors are social and reactive, but they are *not* explicitly required to be "autonomous" in the sense of being proactive [28]. However, autonomous actors may be implemented in AA, and many of our experimental studies require proactive actors. Although the term agent has been used to mean proactive actors, for our purposes the distinction is not critical. In this paper, we use the terms 'agent' and 'actor' as synonyms.

The Actor Architecture consists of two main components:

- *AA platforms* which provide the system environment in which actors exist and interact with other actors. In order to execute actors, each computer node must have one AA platform. AA platforms provide actor state management, actor communication, actor migration, and middle agent services.
- *Actor library* which is a set of APIs that facilitate the development of agents on the AA platforms by providing the user with a high level abstraction of service primitives. At execution time, the actor library works as the interface between actors and their respective AA platforms.

An AA platform consists of eight components (see Fig. 1): Message Manager, Transport Manager, Transport Sender, Transport Receiver, Delayed Message Manager, Actor Manager, Actor Migration Manager, and ATSpace.

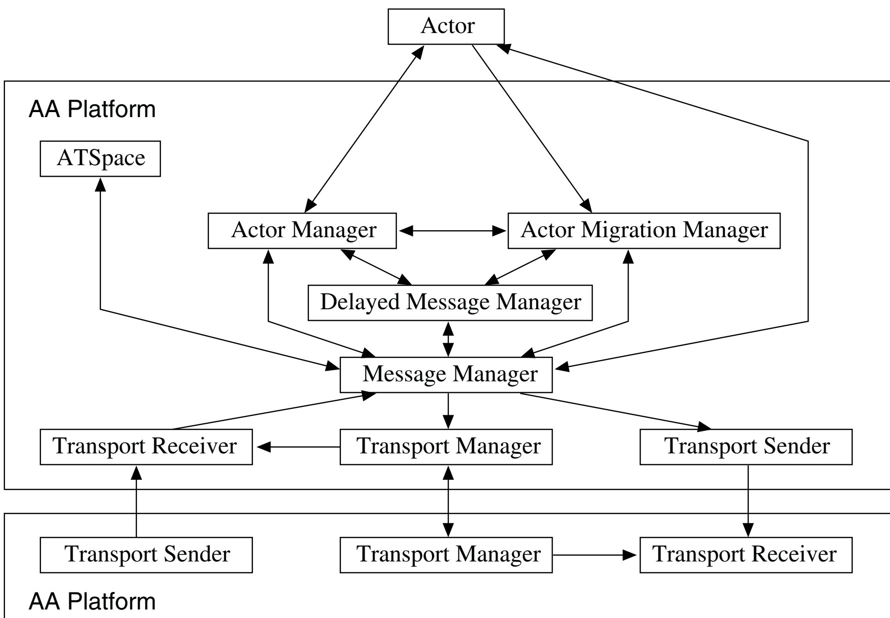


Fig. 1. Architecture of an AA Platform.

The *Message Manager* handles message passing between actors. Every message passes through at least one Message Manager. If the receiver actor of a message exists on the same AA platform, the Message Manager of that platform directly delivers the message to the receiver actor. However, if the receiver actor is not on the same AA platform, this Message Manager delivers the message to the Message Manager of the platform where the receiver currently resides, and finally that Message Manager delivers the message to the receiver actor. The *Transport Manager* maintains a public port for message passing between different AA platforms. When a sender actor sends a message to another actor on a different AA platform, the *Transport Sender* residing on the same platform as the sender receives the message from the Message Manager of that platform and delivers it to the *Transport Receiver* on the AA platform of the receiver. If there is no built-in connection between these two AA platforms, the Transport Sender contacts the Transport Manager of the AA platform of the receiver actor to open a connection so that the Transport Manager can create a Transport Receiver for the new connection. Finally, the Transport Receiver receives the message and delivers it to the Message Manager on the same platform.

The *Delayed Message Manager* temporarily holds messages for mobile actors while they are moving from one AA platform to another. The *Actor Manager* of an AA platform manages the state of actors that are currently executing as well as the locations of mobile actors created on this platform. The *Actor Migration Manager* manages actor migration.

The *ATSpace* provides middle agent services, such as matchmaking and brokering services. Unlike other system components, an ATSpace is implemented as an actor. Therefore, any actor may create an ATSpace, and hence, an AA platform may have more than one ATSpaces. The ATSpace created by an AA platform is called the *default ATSpace* of the platform, and all actors can obtain the names of default ATSpaces. Once an actor has the name of an ATSpace, the actor may send the ATSpace messages in order to use its services for finding other actors that match a given criteria.

In AA, actors are implemented as active objects and are executed as threads; actors on an AA platform are executed with that AA platform as part of one process. Each actor has one actor life cycle state on one AA platform at any time (see Fig. 2). When an actor exists on its original AA platform, its state information appears within only its original AA platform. However, the state of an actor migrated from its original AA platform appears both on its original AA platform and on its current AA platform. When an actor is ready to process a message its state becomes `Active` and stays so while the actor is processing the message. When an actor initiates migration, its state is changed to `Transit`. Once the migration ends and the actor restarts, its state becomes `Active` on the new AA platform and `Remote` on the original AA platform. Following a user request, an actor in the `Active` state may move to the `Suspended` state.

In contrast to other agent life cycle models (e.g. [10, 18]), the AA life cycle model uses the `Remote` state to indicate that an actor that was created on the current AA platform is working on another AA platform.

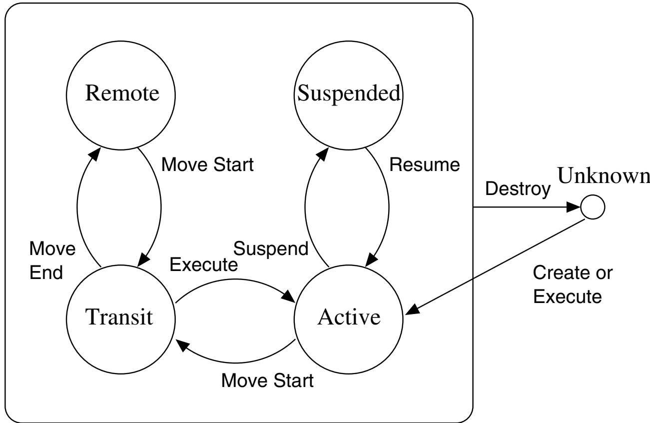


Fig. 2. Actor Life Cycle Model.

3 Optimized Message Delivery

We describe the message delivery mechanisms used to support inter-actor communications. Specifically, AA uses two approaches to reduce the communication overhead for mobile actors that are not on their original AA platforms: *location-based message passing* and *delayed message passing.*

3.1 Location-Based Message Passing

Before an actor can send messages to other actors, it should know the names of the intended receiver actors. In AA, each actor has its own unique name called *UAN* (*Universal Actor Name*). The UAN of an actor includes the *location information* and the *unique identification number* of the actor as follows:

```
uan://128.174.245.49:37
```

From the above name, we can infer that the actor exists on the host whose IP address is `128.174.245.49`, and that the actor is distinguished from other actors on the same platform with its unique identification number `37`.

When the *Message Manager* of a sender actor receives a message whose receiver actor has the above name, it checks whether the receiver actor exists on the same AA platform. If they are on the same AA platform, the Message Manager finds the receiver actor on this AA platform and directly delivers the message. Otherwise, the Message Manager of the sender actor delivers the message to the Message Manager of the receiver actor. In order to find the AA platform where the Message Manager of the receiver actor exists, the location information `128.174.245.49` in the UAN of the receiver actor is used. When the Message Manager on the AA platform with IP address `128.174.245.49` receives the message, it finds the receiver actor there and delivers the message.

The above actor naming and message delivery scheme works correctly when all actors are on their original AA platforms. However, because an actor may

migrate from one AA platform to another, we extend the basic behavior of the Message Manager with a *forwarding* service: when a Message Manager receives a message for an actor that has migrated, it delivers the message to the current AA platform of the mobile actor. To facilitate this service, each AA platform maintains the current locations of actors that were created on it, and updates the location information of actors that have come from other AA platforms on their original AA platforms.

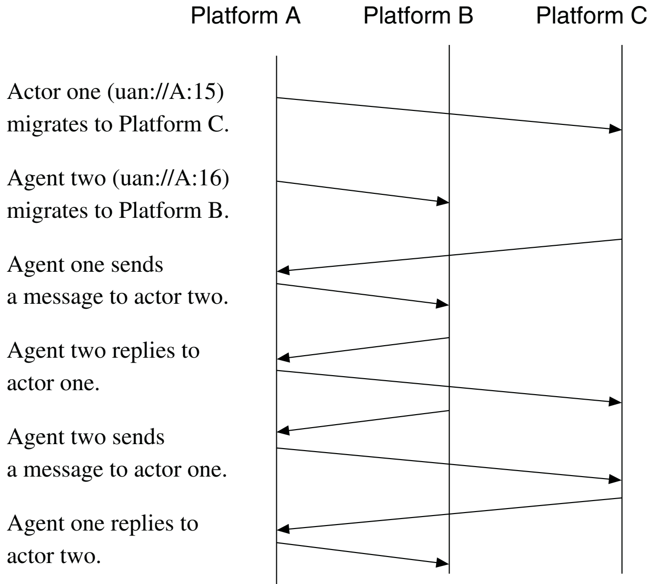
The problem with using only universal actor names for message delivery is that every message for a migrated actor still has to pass through the original AA platform in which the actor was created (Fig. 3.a). This kind of blind indirection may happen even in situations where the receiver actor is currently on an AA platform that is near the AA platform of the sender actor. Since message passing between actor platforms is relatively expensive, AA uses *Location-based Actor Name* (*LAN*) for mobile actors in order to generally eliminate the need for this kind of indirection. Specifically, the LAN of an actor consists of its current location and its UAN as follows:

```
lan://128.174.244.147//128.174.245.49:37
```

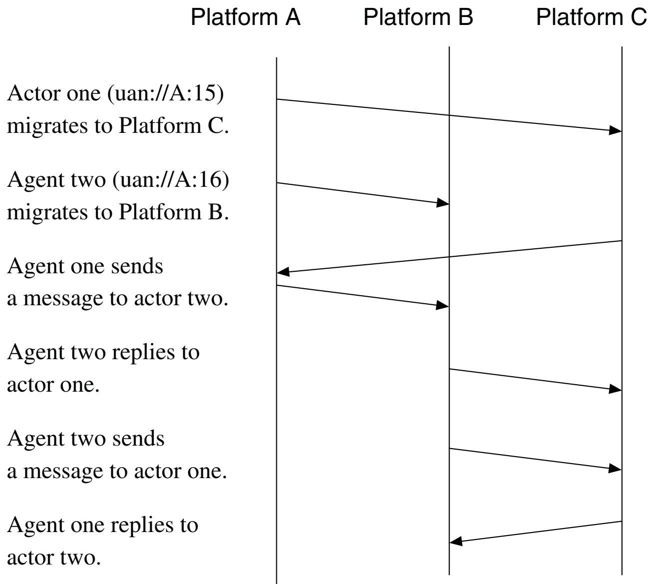
The current location of a mobile actor is set by an AA platform when the actor arrives on the AA platform. If the current location is the same as the location where an actor was created, the LAN of the actor does not have any special information beyond its UAN.

Under the location-based message passing scheme, when the Message Manager of a sender actor receives a message for a remote actor, it extracts the current location of the receiver actor from its LAN and delivers the message to the AA platform where the receiver actor exists. The rest of the procedure for message passing is similar to that in the UAN-based message passing scheme. Fig. 3.b shows how the location-based message passing scheme works. Actor *one* with `ual://C//A:15` sends its first message to actor *two* through the original AA platform of actor *two* because actor *one* does not know the location of actor *two*. This message includes the location information about actor *one* as the sender actor. Therefore, when actor *two* receives the message, it knows the location of actor *one*, and it can now directly send a message to actor *one*. Similarly, when actor *one* receives a message from actor *two*, it learns the location of actor *two*. Finally, the two actors can directly communicate with each other without mediation by their original AA platforms.

In order to use the LAN address scheme, the location information in a LAN should be recent. However, mobile actors may move repeatedly, and a sender actor may have old LANs of mobile actors. Thus a message for a mobile actor may be delivered to its *previous AA platform* from where the actor left. This problem is addressed by having the old AA platform deliver the message to the original AA platform where the actor was created; the original platform always manages the current addresses of its actors. When the receiver actor receives the message delivered through its original AA platform, the actor may send a null



a. UAN−based Message Passing



b. Location−based Message Passing

Fig. 3. Message Passing between Mobile Actors.

message with its LAN to update its location at the sender actor. Therefore, the sender actor can use the updated information for subsequent messages.

3.2 Delayed Message Passing

While a mobile actor is moving from one AA platform to another, the current AA platform of the actor is not well defined. In AA, because the location information of a mobile actor is updated after it finishes migration, its original AA platform thinks the actor still exists on its old AA platform during migration. Therefore, when the Message Manager of the original AA platform receives a message for a mobile actor, it sends the message to the Message Manager of the old AA platform thinking that it is still there. After the Message Manager of the old AA platform receives the message, it forwards the message to the Message Manager of the original AA platform. Thus, a message is continuously passed between these two AA platforms until the mobile actor updates the Actor Manager of its original AA platform with its new location.

In order to avoid unnecessary message thrashing, we use the *Delayed Message Manager* in each AA platform. After the actor starts its migration, the Actor Manager of the old AA platform changes its state to be `Transit`. From this moment, the Delayed Message Manager of this platform holds messages for this mobile actor until the actor reports that its migration has ended. After the mobile actor finishes its migration, its new AA platform sends its old AA platform and its original AA platform a message to inform them that the migration process has ended. When these two AA platforms receive this message, the original AA platform changes the state of the mobile actor from `Transit` to `Remote` while the old AA platform removes all information about the mobile actor, and the Delayed Message Manager of the old AA platform forwards the delayed messages to the Message Manager of the new AA platform of the actor.

4 Active Brokering Service

An ATSpace supports *active brokering services* by allowing agents to send their own search algorithms to be executed in the ATSpace address space [14]. We compare this service to current middle agent services.

Many middle agents are based on *attribute-based communication*. Service agents register themselves with the middle agent by sending a tuple whose attributes describe the service they advertise. To find the desired service agents, a client agent supplies a tuple template with constraints on attributes. The middle agent then tries to find service agents whose registered attributes match the supplied constraints. Systems vary more or less according to the types of constraints (primitives) they support. Typically, a middle agent provides exact matching or regular expression matching [2, 11, 17]. As we mentioned earlier, this solution suffers from a lack of expressiveness and incomplete information.

For example, consider a middle agent with information about seller agents. Each service agent (seller) advertises itself with the following attributes `<actor`

`name, seller city, product name, product price>`. A client agent with the following query is stuck:

Q1: *What are the* ***best two*** *(in terms of price) sellers that offer computers and whose locations are roughly* ***within 50 miles of me****?*

Considering the current tuple space technology, the operator “best two” is clearly not supported (expressiveness problem). Morever, the tuple space does not include distance information between cities (incomplete information problem). Faced with these difficulties, a user with this complex query Q1 has to transform it into a simpler one that is accepted by the middle agent which retrieves a superset of the data to be retrieved by Q1. In our example, a simpler query could be:

Q2: *Find all tuples about sellers that sell computers.*

An apparent disadvantage of the above approach is the movement of a large amount of data from the middle agent space to the buyer agent, especially if Q2 is semantically distant from Q1. In order to reduce communication overhead, ATSpace allows a sender agent to send its own search algorithm to find service agents, and the algorithm is executed in the ATSpace. In our example, the buyer agent would send a search object that would inspect tuples in the middle agent and select the best two sellers that satisfy the buyer criteria.

4.1 Security Issues

Although active brokering services mitigate the limitations of middle agents, such as brokers or matchmakers, they also introduce the following security problems in ATSpaces:

- *Data Integrity*: A search object may not modify tuples owned by other actors.
- *Denial of Service*: A search object may not consume too much processing time or space of an ATSpace, and a client actor may not repeatedly send search objects to overload an ATSpace.
- *Illegal Access*: A search object may not carry out unauthorized accesses or illegal operations.

We address the first problem by preventing the search object from modifying tuple data of other actors. This is done by supplying methods of the search object with a copy of the data in the ATSpace. However, when the number of tuples in the ATSpace is large, this solution requires extra memory and computation resources. Thus the ATSpace supports the option of delivering a shallow copy of the original tuples to the search object at the risk of data being changed by search objects as such scheme may compromise the data integrity.

To prevent malicious objects from exhausting the ATSpace computational resource, we deploy user-level thread scheduling as depicted in Fig. 4. When a search object arrives, the object is executed as a thread and its priority is

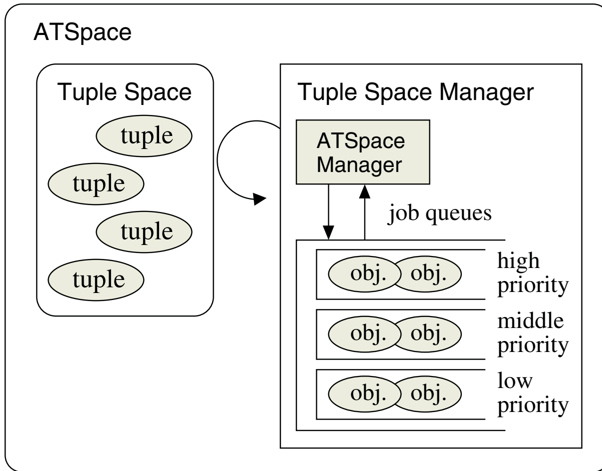


Fig. 4. Architecture of an ATSpace.

set to high. If the thread executes for a long time, its priority is continuously downgraded. Moreover, if the running time of a search object exceeds a certain limit, it may be destroyed by the tuple space manager.

To prevent unauthorized accesses, if the ATSpace is created with an access key, then this key must accompany every message sent from client actors. In this case, actors are allowed only to modify their own tuples. This prevents removal or modification of tuples by unauthorized actors.

5 Experiments and Evaluation

The AA platforms and actors have been implemented in Java language to support operating system independent actor mobility. The Actor Architecture is being used for large-scale UAV (Unmanned Aerial Vehicle) simulations. These simulations investigate the effects of different collaborative behaviors among a large number of micro UAVs during their surveillance missions over a large number of moving targets [15]. For our experiments, we have tested more than 1,000 actors on four computers: 500 micro UAVs, 500 targets, and other simulation purpose actors are executed. The following two sections evaluate our solutions.

5.1 Optimized Message Delivery

According to our experiments, the location-based message passing scheme in AA reduces the number of hops (over AA platforms) that a message for a mobile actor goes through. Since an agent has the location information about its collaborating agents, the agent can carry this information when it moves from one AA platform to another. With location-based message passing, the system is more fault-tolerant; since messages for a mobile actor need not pass through the original AA platform of the actor, the messages may be correctly delivered to the actor even when the actor's original AA platform is not working correctly.

Moreover, delayed message passing removes unnecessary message thrashing for moving agents. When delayed message passing is used, the old AA platform of a mobile actor needs to manage its state information until the actor finishes its migration, and the new platform of the mobile actor needs to report the migration state of the actor to its old AA platforms. In our experiments, this overhead is more than compensated; without delayed message passing the same message may get delivered seven or eight times between the original AA platform and the old AA platform while a mobile actor is moving. If a mobile actor takes more time for its migration, this number may be even greater.

5.2 Active Brokering Service

The performance benefit of ATSpace can be measured by comparing its active brokering services with the data retrieval services of a template-based general middle agent supporting the same service along four different dimensions: the number of messages, the total size of messages, the total size of memory space on the client and middle agent AA platforms, and the computation time for the whole operation. To analytically evaluate ATSpaces, we will use the scenario mentioned in section 4 where a service requesting agent has a complex query that is not supported by the template-based model.

First, with the template-based service, the number of messages is $n+2$ where n is the number of service agents that satisfy a complex query. This is because the service requesting agent has to first send a message to the middle agent to bring a superset of its final result. This costs two messages: a service request message to the middle agent ($\texttt{Service_Request}_{\texttt{template}}$) that contains Q2 and a reply message that contains agent information satisfying Q2 ($\texttt{Service_Reply}_{\texttt{template}}$). Finally, the service requesting agent sends n messages to the service agents that match its original criteria. With the active brokering service, the total number of messages is $n+1$. This is because the service requesting agent need not worry about the complexity of his query and only sends a service request message ($\texttt{Service_Request}_{\texttt{ATSpace}}$) to the ATSpace. This message contains the code that represents its criteria along with the message that should be sent to the agents which satisfy these criteria. The last n messages have the same explanation as in the template-based service.

While the number of messages in the two approaches does not differ that much, the total size of these messages may have a huge difference. In both approaches, a set of n messages needs to be sent to the agents that satisfy the final matching criteria. Therefore, the question of whether or not active brokering services result in bandwidth saving depends on the relative size of the other messages. Specifically the difference in bandwidth consumption (DBC) between the template-based middle agent and the ATSpace is given by the following equation:

$$\begin{aligned} DBC = [&size(\texttt{Service_Request}_{\texttt{template}}) - \\ &size(\texttt{Service_Request}_{\texttt{ATSpace}})] + \\ &size(\texttt{Service_Reply}_{\texttt{template}}) \end{aligned}$$

In general, since the service request message in active brokering services is larger as it has the search object, the first component is negative. Therefore, active brokering services will only result in a bandwidth saving if the increase in the size of its service request message is smaller than the size of the service reply message in the template-based service. This is likely to be true if the original query (Q1) is complex such that turning it into a simpler one (Q2) to retrieve a superset of the result would incur a great semantic loss and as such would retrieve much extra agent information from the middle agent.

Third, the two approaches put a conflicting requirement on the amount of space needed on both the client and middle agent machines. In the template-based approach the client agent needs to provide extra space to store the tuples returned by Q2. On the hand, the ATSpace needs to provide extra space to store copies of tuples given to search objects. However, a compromise can be made here as the creator of the ATSpace can choose to use the shallow copy of tuples.

Fourth, the difference in computation times of the whole operation in the two approaches depends on two factors: the time for sending messages and the time for evaluating queries on tuples. The tuples in the ATSpace are only inspected once by the search object sent by the service requesting agent. However, in the template-based middle agent, some tuples are inspected twice. First, in order to evaluate Q2, the middle agent needs to inspect all the tuples that it has. Second, these tuples that satisfy Q2 are sent back to the service requesting agent to inspect them again and retain only those tuples that satisfy Q1. If Q1 is complex then Q2 will be semantically distant from Q1, which in turns has two ramifications. First, the time to evaluate Q2 against all the tuples in the middle agent is small relative to the time needed to evaluate the search object over them. Second, most of the tuples on the middle agent would pass Q2 and be sent back to be re-evaluated by the service requesting agent. This reevaluation has nearly the same complexity as running the search object code. Thus we conclude that when the original query is complex and external communication cost is high, the active brokering service will result in time saving.

Apart from the above analytical evaluation, we have run a series of experiments on the UAV simulation to substantiate our claims. (Interested readers may refer to [13] for more details.) Fig. 5 demonstrates the saving in computational time of an ATSpace compared to a template-based middle agent that provides data retrieval services with the same semantic. Fig. 6 shows the wall clock time ratio of a template-based middle agent to an ATSpace. In these experiments, UAVs use either active brokering services or data retrieval services to find their neighboring UAVs. In both cases, the middle agent includes information about locations of UAVs and targets. In case of the active brokering service, UAVs send search objects to an ATSpace while the UAVs using data retrieval service send tuple templates. The simulation time for each run is around 35 minutes, and the wall clock time depends on the number of agents. When the number of agents is small, the difference between the two approaches is not significant. However, as the number of agents is increased, the difference becomes large.

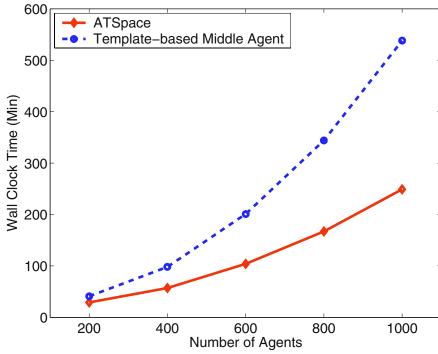


Fig. 5. Wall Clock Time (Min) for ATSpace and Template-based Middle Agent.

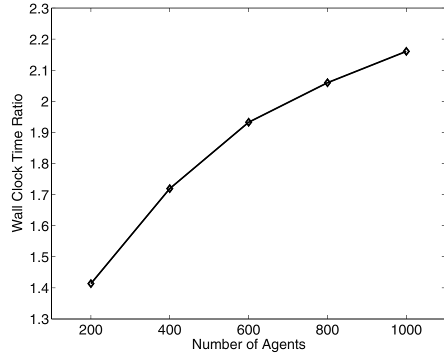


Fig. 6. Wall Clock Time Ratio of Template-based Middle Agent-to-ATSpace.

Fig. 7 depicts the number of messages required in both cases. The number of messages in the two approaches is quite similar but the difference is slightly increased according to the number of agents. Note that the messages increase almost linearly with the number of agents, and that the difference in the number of messages for a template-based middle agent and an ATSpace is small; it is in fact less than 0.01% in our simulations.

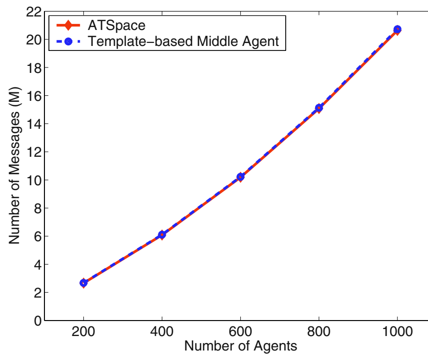


Fig. 7. The Number of Messages for ATSpace and Template-based Middle Agent.

Fig. 8 shows the total message size required in the two approaches, and Fig. 9 shows the total message size ratio. When the search queries are complex, the total message size in the ATSpace approach is much less than that in the template-based middle agent approach. In our UAV simulation, search queries are rather complex and require heavy mathematical calculations, and hence, the ATSpace approach results in a considerable bandwidth saving. It is also interesting to note the relationship between the whole operation time (as shown in Fig. 5) and the bandwidth saving (as shown in Fig. 8). This relationship supports our claim

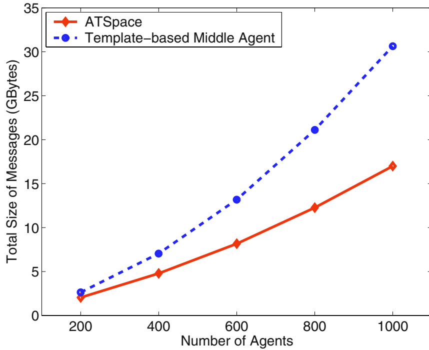


Fig. 8. Total Message Size (GBytes) for ATSpace and Template-based Middle Agent.

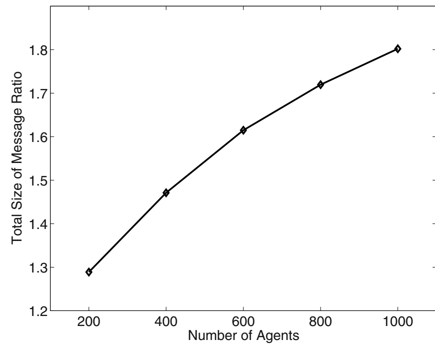


Fig. 9. Total Message Size Ratio for Template-based Middle Agent-to-ATSpace.

that the saving in the total operation time by the ATSpace is largely due to its superiority in efficiently utilizing the bandwidth.

6 Related Work

The basic mechanism of location-based message passing is similar to the message passing in *Mobile IP* [20], although its application domain is different. The original and current AA platforms of a mobile actor correspond to the home and foreign agents of a mobile client in Mobile IP, and the UAN and LAN of a mobile actor are similar to the home address and care-of address of a mobile client in Mobile IP. However, while the sender node in Mobile IP manages a binding cache to map home addresses to care-of addresses, the sender AA platform in AA does not have a mapping table. Another difference is that in mobile IP, the home agent communicates with the sender node to update the binding cache. However, in AA this update can be done by the agent itself when it sends a message that contains its address.

The LAN (Location-based Actor Name) may also be compared to UAL (Universal Actor Locator) in *SALSA* [27]. In SALSA, UAL represents the location of an actor. However, SALSA uses a middle agent called Universal Actor Naming Server to locate the receiver actor. SALSA's approach requires the receiver actor to register its location at a certain middle agent, and the middle agent must manage the mapping table.

The ATSpace approach, which is based on the tuple space model, is related to *Linda* [6]. In the Linda model, processes communicate with other processes through a shared common space called a blackboard or a tuple space without considering references or names of other processes [6, 21]. This approach was used in several agent frameworks, for example *EMAF* [3] and *OAA* [8]. However, these models support only primitive features for pattern-based communication among processes or agents. From the middle agent perspective, *Directory Facilitator* in

the *FIPA* platform [10], *ActorSpace* [2], and *Broker Agent* in *InfoSleuth* [16] are related to our research. However, these systems do not support customizable matching algorithms.

From the expressiveness perspective, some work has been done to extend the matching capability of the basic tuple space model. *Berlinda* [26] allows a concrete entry class to extend the matching function, and *TS* [12] uses policy closures in a Scheme-like language to customize the behavior of tuple spaces. However, these approaches do not allow the matching function to be changed during execution time. At the other hand, *OpenSpaces* [9] provides a mechanism to change matching polices during the execution time. OpenSpaces groups entries in its space into classes and allows each class to have its individual matching algorithm. A manager for each class of entries can change the matching algorithm during execution time. All agents that use entries under a given class are affected by any change to its matching algorithm. This is in contrast to the ATSpace where each agent can supply its own matching algorithm without affecting other agents. Another difference between OpenSpaces and ATSpaces is that the former requires a registration step before putting the new matching algorithm into action, but ATSpace has no such requirement.

Object Space [22] allows distributed applications implemented in the C++ programming language to use a matching function in its template. This matching function is used to check whether an object tuple in the space is matched with the tuple template given in `rd` and `in` operators. However, in the ATSpace the client agent supplied search objects can have a global overview of the tuples stored in the shared space and hence can support global search behavior rather than the one tuple based matching behavior supported in Object Space. For example, using the ATSpace a client agent can find the best ten service agents according to its criteria whereas this behavior cannot be achieved in Object Space.

TuCSoN [19] and *MARS* [5] provide programmable coordination mechanisms for agents through Linda-like tuple spaces to extend the expressive power of tuple spaces. However, they differ in the way they approach the expressiveness problem; while TuCSoN and MARS use reactive tuples to extend the expressive power of tuple spaces, the ATSpace uses search objects to support search algorithms defined by client agents. A reactive tuple handles a certain type of tuples and affects various clients, whereas a search object handles various types of tuples and affects only its creator agent. Therefore, while TuCSoN and MARS extend the general search ability of middle agents, ATSpace supports application agent-oriented searching on middle agents.

Mobile Co-ordination [23] allows agents to move a set of multiple tuple space access primitives to a tuple space for fault tolerance. In *Jada* [7], one primitive may use multiple matching templates. In *ObjectPlaces* [24], dynamic objects are used to change their state whenever corresponding objecplace operations are being called. Although these approaches improve the searching ability of tuple spaces with a set of search templates or dynamic objects, ATSpace provides more flexibility to application agents with their own search code.

7 Conclusion and Future Work

In this papers we addressed two closely related agent communication issues: efficient message delivery and service agent discovery. Efficient message delivery has been addressed using two techniques. First, the agent naming scheme has been extended to include the location information of mobile agents. Second, messages whose destination agent is moving are postponed by the Delayed Message Manager until the agent finishes its migration. For efficient service agent discovery, we have addressed the ATSpace, Active Tuple Space. By allowing application agents to send their customized search algorithms to the ATSpace, application agents may efficiently find service agents. We have synthesized our solutions to the mobile agent addressing and service agent discovery problems in a multi-agent framework.

The long term goal of our research is to build an environment that allows for experimental study of various issues that pertains to message passing and service agent discovery in open multi-agent systems and provide a principled way of studying possible tensions that arise when trying to simultaneously optimize each service. Other future directions include the followings: for efficient message passing, we plan to investigate various trade-offs in using different message passing schemes for different situations. We also plan to extend the Delayed Message Manager to support mobile agents who are contiguously moving between nodes. For service agent discovery, we plan to elaborate on our solutions to the security issues introduced with active brokering services.

Acknowledgements

The authors would like to thank the anonymous reviewers and Naqeeb Abbasi for their helpful comments and suggestions. This research is sponsored by the Defense Advanced Research Projects Agency under contract number F30602-00-2-0586.

References

1. G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.
2. G. Agha and C.J. Callsen. ActorSpaces: An Open Distributed Programming Paradigm. In *Proceedings of the 4th ACM Symposium on Principles and Practice of Parallel Programming*, pages 23–32, May 1993.
3. S. Baeg, S. Park, J. Choi, M. Jang, and Y. Lim. Cooperation in Multiagent Systems. In *Intelligent Computer Communications (ICC '95)*, pages 1–12, Cluj-Napoca, Romania, June 1995.
4. F. Bellifemine, A. Poggi, and G. Rimassa. JADE - A FIPA-compliant Agent Framework. In *Proceedings of Practical Application of Intelligent Agents and Multi-Agents (PAAM '99)*, pages 97–108, London, UK, April 1999.
5. G. Cabri, L. Leonardi, and F. Zambonelli. MARS: a Programmable Coordination Architecture for Mobile Agents. *IEEE Computing*, 4(4):26–35, 2000.

6. N. Carreiro and D. Gelernter. Linda in Context. *Communications of the ACM*, 32(4):444–458, 1989.
7. P. Ciancarini and D. Rossi. Coordinating Java Agents over the WWW. *World Wide Web*, 1(2):87–99, 1998.
8. P.R. Cohen, A.J. Cheyer, M. Wang, and S. Baeg. An Open Agent Architecture. In *AAAI Spring Symposium*, pages 1–8, March 1994.
9. S. Ducasse, T. Hofmann, and O. Nierstrasz. OpenSpaces: An Object-Oriented Framework for Reconfigurable Coordination Spaces. In A. Porto and G.C. Roman, editors, *Coordination Languages and Models, LNCS 1906*, pages 1–19, Limassol, Cyprus, September 2000.
10. Foundation for Intelligent Physical Agents. *SC00023J: FIPA Agent Management Specification*, December 2002. http://www.fipa.org/specs/fipa00023/.
11. N. Jacobs and R. Shea. The Role of Java in InfoSleuth: Agent-based Exploitation of Heterogeneous Information Resources. In *Proceedings of Intranet-96 Java Developers Conference*, April 1996.
12. S. Jagannathan. Customization of First-Class Tuple-Spaces in a Higher-Order Language. In *Proceedings of the Conference on Parallel Architectures and Languages - Vol. 2, LNCS 506*, pages 254–276. Springer-Verlag, 1991.
13. M. Jang, A. Ahmed, and G. Agha. A Flexible Coordination Framework for Application-Oriented Matchmaking and Brokering Services. Technical Report UIUCDCS-R-2004-2430, Department of Computer Science, University of Illinois at Urbana-Champaign, 2004.
14. M. Jang, A. Abdel Momen, and G. Agha. ATSpace: A Middle Agent to Support Application-Oriented Matchmaking and Brokering Services. In *IEEE/WIC/ACM IAT(Intelligent Agent Technology)-2004*, pages 393–396, Beijing, China, September 20-24 2004.
15. M. Jang, S. Reddy, P. Tosic, L. Chen, and G. Agha. An Actor-based Simulation for Studying UAV Coordination. In *15th European Simulation Symposium (ESS 2003)*, pages 593–601, Delft, The Netherlnds, October 26-29 2003.
16. R.J. Bayardo Jr., W. Bohrer, R. Brice, A. Cichocki, J. Fowler, A. Helal, V. Kashyap, T. Ksiezyk, G. Martin, M. Nodine, M. Rashid, M. Rusinkiewicz, R. Shea, C. Unnikrishnan, A. Unruh, and D. Woelk. InfoSleuth: Agent-Based Semantic Integration of Information in Open and Dynamic Environments. *ACM SIGMOD Record*, 26(2):195–206, June 1997.
17. D.L. Martin, H. Oohama, D. Moran, and A. Cheyer. Information Brokering in an Agent Architecture. In *Proceedings of the Second International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology*, pages 467–489, London, April 1997.
18. D.G.A. Mobach, B.J. Overeinder, N.J.E. Wijngaards, and F.M.T. Brazier. Managing Agent Life Cycles in Open Distributed Systems. In *Proceedings of the 2003 ACM symposium on Applied Computing*, pages 61–65, Melbourne, Florida, 2003.
19. A. Omicini and F. Zambonelli. TuCSoN: a Coordination Model for Mobile Information Agents. In *Proceedings of the 1st Workshop on Innovative Internet Information Systems*, Pisa, Italy, June 1998.
20. C.E. Perkins. Mobile IP. *IEEE Communications Magazine*, 35:84–99, May 1997.
21. K. Pfleger and B. Hayes-Roth. An Introduction to Blackboard-Style Systems Organization. Technical Report KSL-98-03, Stanford Knowledge Systems Laboratory, January 1998.
22. A. Polze. Using the Object Space: a Distributed Parallel make. In *Proceedings of the 4th IEEE Workshop on Future Trends of Distributed Computing Systems*, pages 234–239, Lisbon, September 1993.

23. A. Rowstron. Mobile Co-ordination: Providing Fault Tolerance in Tuple Space Based Co-ordination Languages. In *Proceedings of the Third International Conference on Coordination Languages and Models*, pages 196–210, 1999.
24. K. Schelfthout and T. Holvoet. ObjectPlaces: An Environment for Situated Multi-Agent Systems. In *Third International Joint Conference on Autonomous Agents and Multiagent Systems - Volume 3 (AAMAS'04)*, pages 1500–1501, New York City, New York, July 2004.
25. K. Sycara, K. Decker, and M. Williamson. Middle-Agents for the Internet. In *Proceedings of the 15th Joint Conference on Artificial Intelligences (IJCAI-97)*, pages 578–583, 1997.
26. R. Tolksdorf. Berlinda: An Object-oriented Platform for Implementing Coordination Language in Java. In *Proceedings of COORDINATION '97 (Coordination Languages and Models), LNCS 1282*, pages 430–433. Pringer-Verlag, 1997.
27. C.A. Varela and G. Agha. Programming Dynamically Reconfigurable Open Systems with SALSA. *ACM SIGPLAN Notices: OOPSLA 2001 Intriguing Technology Track*, 36(12):20–34, December 2001.
28. M. Wooldridge. *An Introduction to MultiAgent Systems.* John Wiley & Sons, Ltd, 2002.

Adaptive Access Control in Coordination-Based Mobile Agent Systems

Christine Julien[1], Jamie Payton[2], and Gruia-Catalin Roman[2]

[1] Department of Electrical and Computer Engineering
The University of Texas at Austin
c.julien@mail.utexas.edu

[2] Department of Computer Science and Engineering
Washington University in Saint Louis
{payton,roman}@wustl.edu

Abstract. The increased pervasiveness of mobile devices like cell phones, PDAs, and laptops draws attention to the need for coordination among these networked devices. The very nature of the environment requires devices to interact opportunistically when resources are available. Such interactions occur unpredictably as device users have no advance knowledge of others they will encounter. The openness of these environments also requires users to protect themselves and their data from unwanted interactions while maintaining desired, yet unscripted, coordination. As the ubiquity of communicating mobile devices increases, the number of applications supported by the network grows drastically and managing access control is crucial to such systems. Application agents must directly manipulate and examine access policies because these networks are often decoupled from a fixed infrastructure, rendering reliance on centralized servers for authentication and access policies impractical. In this paper, we explore context-aware access control policies tailored to the needs of agent coordination in open environments that exhibit mobility. We propose and evaluate novel constructs to support such policies, especially in the presence of large numbers of highly dynamic application agents.

1 Introduction

Ubiquitous computing devices communicate wirelessly, opportunistically forming ad hoc networks not connected to a wired infrastructure. These networks can include a handful of devices or thousands of heterogeneous components, making coordinating and mediating their competing needs a massive task. In such environments, distributed applications exchange information or coordinate tasks. These applications are commonly structured as logical networks of mobile agents. Mobile agents (or application agents) carry all or part of a particular application's behavior and are empowered with the ability to move through the network of physically mobile devices. Much research focuses on developing middleware to facilitate interactions among these highly dynamic application agents.

This paper focuses on systems that use tuple spaces for coordination, The original Linda model [1] provides a centralized tuple space where application

R. Choren et al. (Eds.): SELMAS 2004, LNCS 3390, pp. 254–271, 2005.
© Springer-Verlag Berlin Heidelberg 2005

agents exchange information using content-based matching of patterns against data. Variations on this theme adapt it to the mobile environment where a central repository is not feasible. The benefits of a tuple space model are twofold. First, the tuple space affords a decoupled manner of communication, eliminating the need for a priori knowledge of the identities of communication partners. This facilitates flexible coordination in open environments in which mobile agents come and go without notice. Second, the model masks the complex communication details associated with handling frequent, unannounced disconnections that characterize mobile networks. This allows novice programmers to create complex applications in environments for which it is generally difficult to program.

Tuple space implementations have enjoyed much popularity not only within the research community, but also in the commercial sector, where applications have reached real-world deployed status. OptimalGrid [2] uses IBM's TSpaces [3] to coordinate parallel processes in large-scale computations. TSpaces also supports communication among devices in an automobile, among components of a smart house, and in vending machine maintenance. JavaSpaces [4] supports the Jini service infrastructure and has been deployed in many situations including the integration of proprietary law enforcement databases to enhance information availability and the creation of tourism networks linking potential travelers, airlines, and hotels. More recently, a number of mobile agent middleware systems designed for ad hoc networks have begun to utilize tuple space based coordination including Lime [5], EgoSpaces [6], and MARS [7]. These systems address tuple space coordination in highly dynamic environments.

In open and dynamic mobile systems, security concerns of three types arise: protecting hosts from malicious agents, protecting agents from tampering hosts, and securing data. Commonly referenced approaches [8] address the first two concerns in mobile agent systems. Executing agents using "safe interpreters" [9–11] provides a sandboxing effect that protects hosts from errant code. Proof-carrying code [12] can verify an agent before it runs on a new host. D'Agents [10] uses public-key cryptography to authenticate incoming agents. The more difficult problem of protecting agents from tampering hosts comes in two forms: detecting a malicious event and preventing the leakage of sensitive information. The former can be accomplished by examining execution traces while encryption schemes [13] have helped to preserve an agent's secrecy. Finally, undetachable threshold signatures [14] prevent hosts from tampering with an agent's data.

Protecting data includes ensuring secrecy and controlling data access. Much research in ad hoc networks has specifically addressed securing ad hoc routing protocols. In addition, approaches like the Secure Message Transmission protocol [15] focus on protecting individual data transmissions. Even within the coordination arena, researchers have devised encryption schemes for communication with coordination spaces. For example, SAMCat [16] and Yalta [17] use encryption and authentication to securely transmit tuples into and out of a data space. Our work focuses on the final issue: controlling access to data. A solution to this problem is complicated by the fact that, in the mobile environment, disconnection from a wired infrastructure renders a centralized solution impossible.

In traditional access control solutions, a single administrator determines what kind of access can be provided to particular subjects for certain objects. A common mechanism in wired networks uses access matrices to describe rights. The rows of the matrix correspond to users and the columns to objects; a cell in the matrix contains the access rights a user has on an object. This approach generalizes several approaches, including access control lists and capability definitions. In the mobile environment, the number of possible agents and the amount of data available over the lifetime of the system make direct application of these solutions impractical. The access control function introduced in this paper overcomes the limitations imposed by mobile systems by operating over general descriptions of interacting parties and dynamically adjusting to the changing context.

Section 2 introduces a general coordination model for mobile computing. Section 3 describes our access control mechanism. Details of a particular implementation of this mechanism appear in Section 4 and applications showing its use in Section 5. In Section 6, we discuss the construct's expressive power and overhead. Section 7 overviews related work, and conclusions appear in Section 8.

2 A Generalized Coordination Model

In this section, we capture the essential features of tuple space coordination mechanisms in mobile agent systems. This generalization of coordination allows us to focus our efforts on creating access control that is not tailored for use in a specific system. The result is a generalization that spans the gamut from tuple definition to sophisticated tuple space operations.

2.1 Linda Tuple Space Model

Linda enables coordination through the use of a centralized data repository. Processes insert data by generating tuples in the repository and retrieve data through content-based operations on the tuple space. In such an operation, the requesting process specifies a pattern that the retrieved tuple must match. These operations are synchronous in that they "block" the issuing process until a tuple satisfies the operation. Adaptations of this model of coordination have proven useful in mediating interactions among components that require decoupling in both space and time, a characteristic of highly dynamic or mobile systems.

2.2 Computational Model

We assume a computing model in which devices (or *hosts*) can move in physical space and applications are structured as a community of mobile agents that can migrate among hosts. In our computing model, the agent is the unit of modularity, execution, and mobility, while a host is a container for agents characterized by, among other things, a location in physical space. We use the term agent to refer to any stand-alone piece of software code capable of moving between connected hosts. Communication among agents and agent migration can take place whenever the hosts involved can physically communicate with each other.

2.3 The Tuple Space

Some mobile systems (e.g., MARS [7]) focus on logically mobile agents in a network of physically stationary hosts, while other systems (e.g., LIME [5] and EgoSpaces [6]) integrate physical and logical mobility. Each of these systems facilitates interactions among large numbers of application agents by using a tuple space that other hosts or agents can access. Tuple spaces can be permanently bound to hosts, to agents, or distributed among a combination of the two. The distribution of the tuples is irrelevant with respect to access control; the key aspect of the representation is how application agents access data. In this paper, we assume a tuple space bound to each mobile agent. This choice is motivated simply from a modeling perspective to simplify the discussion of and reasoning about our access control policies. Using this model, we can simulate other approaches. For example, to simulate tuple spaces bound to a host, we permanently associate an agent to each host and use its tuple space as the host's tuple space. On the other hand, to simulate access control policies bound to an individual data item, we can create a new agent for the individual data item. The data item's access control is then controlled by the dedicated agent.

The control of each unit (agent, host, or event data item) over its own data caters to the needs of mobile applications that must often operate autonomously in order to handle the uncertainty of the environment. Agents or devices may interact for a period of time, only to be disconnected and never meet again. Such challenges render any centralized approach to data management infeasible.

2.4 Tuples and Patterns

We generalize a tuple to one in which each field is identified by a name. A tuple is an unordered set of triples: $\langle(name, type, value), \ldots\rangle$. For each field, *type* is the data type of *value*. In a tuple, each field *name* must be unique. Users access tuple spaces by matching patterns against tuples. A pattern has the form: $\langle(name, type, constraint), \ldots\rangle$. A *constraint* provides requirements a field's *value* must match for the tuple's field to match the pattern's field. Specifically, the matching function $\mathcal{M}$ is defined over a tuple θ and pattern p as:

$$\begin{aligned}\mathcal{M}(\theta, p) \equiv \langle \forall c : c \in p :: \langle \exists f : f \in \theta \wedge f.name = c.name \\ \wedge f.type \ \mathtt{instanceof}\ c.type \\ :: c.constraint(f.value)\rangle\rangle.\end{aligned}$$ [1]

$\mathcal{M}$ requires that, for every constraint in the pattern, there is a field in the tuple with the same name, the same type (or a derived type), and a value that satisfies the constraint. While the function requires that each constraint is satisfied, it does not require that every field in the tuple is constrained, i.e., a tuple must contain all the fields in the pattern but can contain additional fields.

[1] In the notation $\langle \mathbf{op}\ quantified_vars : range :: exp\rangle$, the variables from *quantified_vars* take on all values permitted by *range*. Each instantiation of the variables is substituted in *exp*, producing a multiset of values to which **op** is applied, yielding the value of the three-part expression. If no instantiation of the variables satisfies *range*, the value of the expression is the identity element for **op**, e.g., *true* when **op** is $\forall$.

2.5 Basic Operations

Next, we classify the available operations, regardless of the tuple space structure. These operations fall into two categories: tuple generation and tuple retrieval. The former create new data items that agents can share for coordination purposes, while the latter allow agents to access available data items.

Tuple Generation. Agents create tuples using **out** operations: **out**(T, t), where T is a tuple space with a particular name located at a particular agent, and t is a tuple placed in T. In EgoSpaces, an **out** places the tuple in a local tuple space controlled by the generating agent. In LIME an **out** can place a tuple in any tuple space owned by any agent on a connected host. In MARS the tuple is created in the local host's tuple space. For the purposes of access control, understanding tuple generation is important if agents can create tuples in other agents' tuple spaces. In these cases, the agent responsible for the target tuple space often desires the ability to express restrictions on the types of data that can be inserted or on which other agents can generate that data.

Tuple Retrieval. To read and remove tuples, agents use **rd** and **in** operations respectively, which assume three forms: blocking, atomic probing, and scattered probing. The blocking form, **rd**(T, p), returns a tuple matching the pattern p from the tuple space T. The tuple space can be either local to the agent or controlled by another agent. Atomic probing operations, **rdp** and **inp**, guarantee, if a matching tuple exists, it is returned, but they can return ϵ if no match exists. Like the blocking operations, they are atomic with respect to the tuple space on which they are issued; in some cases in the mobile environment, guaranteeing this atomicity can be expensive. Scattered probing operations, **rdsp** and **insp** offer weaker guarantees. While these access operations entail only single tuples, many extensions allow simultaneous access to groups of tuples. These operations come in all three forms described above and are referred to as group operations, e.g., **rdg** refers to a blocking operation that returns all matching tuples from the tuple space. Access control for tuple retrieval operations is more obvious and natural than for the former tuple generation operations. The agent in control of the data items may desire some data to be read only, visible only to certain parties, or mutable only under certain conditions.

Different models present tuple space operations to the user in different ways. In LIME, agents operate over a federation of connected tuple spaces, while in EgoSpaces, agents operate over projections, called *views*, of all available data. These complex interactions can be reduced to the operations described above. We next investigate providing access control mechanisms for systems whose interactions can be expressed using this generalized tuple space model.

3 Access Control Function

Given the coordination model described previously, an agent assumes responsibility for mediating access to its data. The ability to control access in this manner is fundamental because it allows the access policies to reflect an agent's instantaneous needs. This is especially important in the highly dynamic mobile

environment where mobile agents want to constantly adjust their behavior to adapt to a changing context that can include communicating with unpredictable parties. To achieve flexible access control in this environment, each agent specifies an individualized access control function.

We allow an agent to restrict which other agents access its data and the manner in which the access occurs. To accomplish the former, a requesting agent must provide credentials identifying itself. To accomplish the latter, the access control function accounts for the operation being performed. In the end, each agent defines a single access control function that takes as parameters a tuple, a set of credentials identifying the requesting agent, the operation being performed, the pattern used in the operation, and the owning agent's profile (defined next). This function returns a boolean indicating whether the requested access is allowed.

3.1 Profiles

We introduce a profile to maintain properties of each agent, which we represent as a tuple. Particular applications or coordination systems may require specific attributes in this profile. In general, we assume a profile contains at least a unique host id identifying the agent's host and a unique agent id.

3.2 Parameters

An access control function takes five parameters: the credentials, operation, tuple, pattern, and the owner's profile. We limit ourselves to these parameters because they capture the aspects of the coordination model we outlined previously. One could envision the inclusion of additional parameters that measure behaviors over the lifetime of the system, e.g., an access decision could be made based on the history of operations on a particular data item. We choose not to include those at this time because we feel the required bookkeeping overhead is not met by a demand from potential applications.

Credentials. Credentials allow an agent to convey information about itself. In simple cases, they can be a standard set of attributes, e.g., the agent's id or a third-party authentication. When an agent has a priori knowledge of the access requirements, credentials can be more complicated, e.g., a password. When constructing credentials, an agent may desire not to give away too much information, e.g., if the agent has multiple passwords, it should send only the correct one. However, this is not required in our access control mechanism because an agent's credentials are not directly exposed to other agents. These expressive credentials are especially beneficial in open and dynamic mobile environments, where it is often not possible to know a priori which agents can access restricted information. Instead, agents must prove they have required privileges. Agents select their credentials from the union of the host profile and the agent profile. The credentials are then presented as a tuple of attributes, which allows an access control function to use pattern matching to evaluate credentials. The credentials and their transmission with the operation are assumed to be private. This security is outside the scope of this paper but could be accomplished using cryptography schemes already under development.

Operation. The access control function can also account for the operation requested. Often, some data should be restricted to read-only access, yet current systems do not inherently allow this restriction. Considering the operation when determining access allows a dynamic application to permit one set of operations for some agents, but different operations for others.

Requested Tuple. The access control function can operate over the tuple to be returned from an operation. Pattern-matching allows this portion of the access control function to be easily defined while remaining flexible.

Pattern. A powerful component of the access control function is its ability to account for the pattern used in the content-based operation. The pattern provides information about an application's prior knowledge of the data. The owning agent may allow access only to agents that know the "correct" way to access the data (e.g., providing a wild card pattern that matches any tuple may not be acceptable). Some knowledge of the structure of the requested tuple might indicate that the requesting agent shares common application goals.

Owner's Profile. The access control function also considers the owner's current state. Because the access policy is determined dynamically, access can be granted based on context information. In some cases, data may never be sent wirelessly between devices unless they are within a secure physical environment where eavesdropping is known to be impossible.

3.3 The Access Control Function Defined

Formally, the access control function can be represented as: $\mathrm{ACF} : T \times C \times O \times P \times \Pi \rightarrow \{0, 1\}$, where T is the universe of tuples, C is the universe of credentials, O is the finite set of operations, P is the universe of patterns, and Π is the universe of profiles. The access control function (ACF) maps the values of the parameters to a boolean indicating the access decision. The function can also be represented as: $access = \mathrm{ACF}(credentials_r, op, tuple, pattern, profile_o)$; r is the requesting agent and o is the tuple's owner.

We discuss the expressive power of this construct later. For now we consider what it *cannot* easily represent. Access decisions cannot be based on properties of the requesting agent not included in its credentials. Therefore the requesting agent must carefully construct the credentials it sends with each request. The access decision cannot rely on arbitrary environmental properties, e.g., an agent cannot base a decision on the number of copies of a tuple. The access control function lends itself well to mobile environments because it allows adaptive policies. Access decisions are transparent to requesting agents; if access is denied, a requester does not even know that the matching tuple existed.

4 A Sample Implementation

The access control model is intentionally not presented in the context of any particular system. Instead, we have argued that it can be integrated with many tuple space based coordination systems matching the form described in Section 2. As

a demonstration of the feasibility and mechanics of such an integration, we have added this access control mechanism to a particular coordination middleware, EgoSpaces. We expect that, while some of the challenges we encountered are unique, other lessons learned will apply across coordination models.

In this section, we first highlight the novel features of EgoSpaces that make it amenable to coordination in ad hoc networks. This discussion also provides the information necessary to understand the integration of our access control mechanism. We complete this section with a technical description of the implementation of the access control mechanism within EgoSpaces. The description of the EgoSpaces model and middleware is intentionally brief. The interested reader can find a more careful evaluation of the model and its associated research concerns in the literature [6].

4.1 EgoSpaces Overview

EgoSpaces addresses the needs of agents in large-scale heterogeneous environments. An agent operates over a context that can include, in principle, all data in an entire ad hoc network. EgoSpaces' unique model of coordination, however, structures data in terms of *views*, or projections of the maximal set of data. Each agent defines its own views; these individual views abstract the dynamic environment by constraining properties of the network, hosts, agents, and data. To further reduce programming costs, EgoSpaces transparently maintains views; as hosts and agents move, a view's content automatically reflects the context changes without the agent's explicit intervention.

Practically, an agent defines its view as a set of constraints over the network, hosts, agents, and data. Within EgoSpaces each view is managed by an `EgoManager`. Each host is associated with a single `EgoManager`, and all the agents residing on a host register with the `EgoManager` before coordinating with other agents. When registering, an agent's local tuple space contents become the responsibility of the `EgoManager`, who mediates communication between connected agents. The application agents implicitly use the `EgoManager` to define and interact with their views, which can require the `EgoManager` to interact with other `EgoManagers` (and, by association, other agents) on remote hosts. An agent issues content-based retrieval operations on its views. These operations are actually serviced by the `EgoManager` with which the agent is registered. The `EgoManager` uses the pattern provided to select tuples that match the operation request and evaluates each tuple individually to determine whether or not the tuple satisfies the view and is a viable candidate for return to the requesting agent.

4.2 Integrating Access Controls with EgoSpaces

EgoSpaces employs the agent-specified access control function on a per-view basis. When an agent defines a view, it attaches a set of credentials and a list of operations it intends to perform on the view. The EgoSpaces middleware can then use each contributing agent's access control function to determine which tuples belong in the view. In the end, the view contains only the tuples that qualify via their owning agent's access control function.

In providing access controls in EgoSpaces, we use credentials and access control functions along with the content-based retrieval and pattern matching mechanism already present in the system. Upon integrating the access control function, a set of credentials is now included as part of the view definition. These credentials are simply properties that convey information about the agent. The agent can alter its credentials at any time. To restrict other agents' perspectives according to their respective credentials, each agent also provides a dynamically modifiable access control function. A requesting agent's credentials are compared to the access control function of agents who contribute data to the view to restrict the tuples available in the view. With the access control functions in place, to evaluate a tuple for return to a requested operation an `EgoManager` extracts information about the agent (properties of the host the agent resides on, properties of the agent, and the agent's access control function) providing the tuple and compares this information with the constraints defined in the requesting agent's view, *including the credentials*. The latter is the key to the access control function's integration into the EgoSpaces middleware. If the tuple satisfies the view's constraints *and* the requesting agent's credentials satisfy the tuple owner's access control function, then the requested operation can be performed.

An important aspect of the integration of the access control mechanism described in Section 3 into EgoSpaces revolved around the fact that it relies on the mechanisms inherent to tuple space based systems. Tuples are used to describe credentials, and access control functions can be described by a set of access policies defined as patterns, or templates, over tuples. Implementing credentials and access control functions in this way provides a number of benefits. First, the pattern matching mechanisms already provided by the tuple space system can be used to check the credentials against an access control function. Second, we allow the programmer to construct credentials and access control functions in a way that he is already familiar with. Third, using tuples and templates allows for flexibility and adaptation, since adding and removing fields from tuples and patterns is relatively simple. Finally, the use of tuples and patterns allows for expressive access control functions and credentials since access control may be expressed according to any property of the interacting agents.

The EgoSpaces system requires certain assumptions about its operating environment to provide atomic consistency guarantees regarding the performance of its operations on views. More details on these assumptions and dealing with environments where they do not hold can be found in [6]. Because the added access control provisions involve only local decisions at each contributing host, they have no negative impact on view consistency.

5 Programming with Access Control Mechanisms

In this section we demonstrate the use of access controls within the framework of the EgoSpaces coordination system. We first describe the programming interface for using the access control function within EgoSpaces. We then describe two specific applications that use the described interfaces. We selected examples

that apply in differing application domains to give a sense of the access control mechanism's flexibility. We do not give extensive details of the coordination mechanics specific to the EgoSpaces middleware but instead focus on the access control aspects of the two applications.

5.1 The Access Control API

Figure 1 shows the public API for defining and using credentials. As discussed in the previous section, an agent defines credentials that it sends with its view definition in EgoSpaces (or simple operations in other coordination systems) to identify itself to the other party. The first method in the Credentials interface, **selectProperty**, allows the agent to select a property from either the agent's own profile or its host profile to include in the credentials. The second method, **dropProperty**, allows an agent to remove a property from its credentials.

selectProperty(String profileType, String propertyName)
– select a property (identified by the propertyName) from either the host or agent profile (identified by profileType) to include in the credentials
dropProperty(String propertyName)
– drop the property identified by propertyName from the credentials

Fig. 1. The Credentials API Within the EgoSpaces Middleware.

addConstraint(String property)
– add a constraint that requires the existence of field with the given name
addConstraint(String property, String function, Serializable value)
– add a constraint that requires the named field to satisfy the given function and value
addConstraint(String property, ConstraintFunction cf)
– add a constraint that requires the named field to satisfy the given application-defined constraint function
addPermittedOperation(String operation)
– add the specified operation or operations to the list of those allowed
matches(Credentials)
– determines whether the provided credentials match this policy

Fig. 2. The AccessControlPolicy API Within the EgoSpaces Middleware.

An agent who provides data defines an access control function to protect itself and its data. Each access control function is composed of one or more access control policies. The API for this component appears in Figure 2. The API contains three mechanisms to add a constraint to the access control policy. The first allows an agent to add a constraint that requires the credentials to contain a field with a certain name but no specific value. The second mechanism allows the agent to add a constraint that uses a built-in function (e.g., "=") to constrain the value of a named property in the credentials. The third and final

mechanism allows the application agent to define a tailored constraint function that restricts the value of the named property in the credentials. This API also shows the method an agent uses to restrict the operations that can be performed on the coordination space. The final method evaluates the provided credentials to determine whether they match the constraints of this policy.

We provide the access control function as a disjunction of access control policies to allow more expressive functions. We require the combination of the credentials and the specific operation to satisfy at least one of the policies within the access control function. Figure 3 shows how agents assemble access control policies into a single access control function through an add method and a remove method. The **matches** method determines whether the credentials and the operation satisfy at least one of the access control policies.

addPolicy(AccessControlPolicy acp)
– add the specified policy to the agent's access control function
removePolicy(AccessControlPolicy acp)
– remove the specified policy from the agent's access control function
matches(Credentials cred)
– determines whether or not the provided credentials match the policies contained within this access control function

Fig. 3. The AccessControlFunction API Within the EgoSpaces Middleware.

5.2 A Music Sharing Application

A music sharing application for mobile users implemented on top of EgoSpaces serves as one vehicle for testing the access control implementation. The application provides users with access to a music service with sharing, search, and down-load capabilities. To determine what music a user sees, the user provides properties that define the music sharing application's view. This includes a network constraint that includes only data residing on hosts within a certain number of network hops, a host constraint that requires the data to reside on hosts which are traveling in the same direction as the user, and a data constraint that restricts the returned items according to a file size limit. A screen shot of the resulting application is shown in Figure 4.

The data is also restricted according to the credentials provided by the agent, which includes a unique agent id and a known phrase encrypted with a shared password provided in the user's official registration from the music service. This password encrypted phrase authenticates the user as a subscriber. This phrase is provided as a product key when the user retrieves (purchases) the application from a reputable source (vendor). Since users share music only with others subscribed to the service, the agent also provides an access control policy which specifies that a requesting agent must have an agent id and must have the correct phrase encrypted with the subscription password. Successful decryption of the phrase by the receiving agent implies that the requesting agent holds the correct password. The code to define the credentials within the application is:

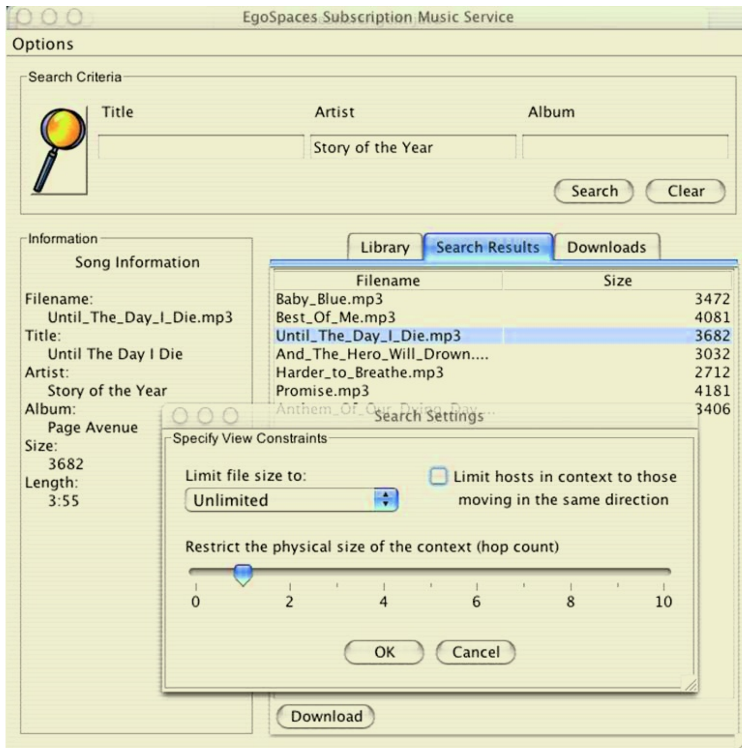


Fig. 4. The subscription music service.

```
Credentials c = new Credentials();
c.selectProperty(AGENTPROFILE, ‘‘Passphrase’’);
```

First, the agent creates a credentials object. It then selects the passphrase property from the agent's profile that was handed out when the code was installed.

To build the access control policy, the agent defines the policy and adds it to the access control function:

```
AccessControlPolicy policy = new AccessControlPolicy();
policy.addConstraint(‘‘Passphrase’’, ‘‘=’’, encryptedPhrase);
policy.addPermittedOperation(Operations.ALLRDS);
acf.addPolicy(policy);
```

In this code, the agent first creates a new policy. It then adds the single constraint that requires the passphrase to be equivalent to this agent's known encrypted phrase. It then adds to the permitted operations all read operations, preventing any admitted agents from removing any of the agent's own music files. Finally, the agent adds the defined policy to the access control function.

This music sharing application requires an initialization which can be arguably termed centralized. As indicated above, it can be equated with receiving the software with a subscription, from a reputable source which provides the

appropriate product key. After installing the music sharing software, users share music in a completely decentralized fashion, making autonomous decisions with no reliance on the availability of a centralized authority.

5.3 Administrative Domains

Many applications restrict agent operations to administrative domains. Assume nested domains defined as a university, a department, and a research group. To provide security guarantees, applications limit access to certain data to only computers on the university's network. Still other data ought to be restricted to departmental computers or to research group computers. A user in the research group, working on a mobile computer, wants to use a software license of which the research group has n copies. The licenses are stored as tuples in a tuple space. Each computer in the group carries a tuple space; the available licenses are initially distributed in some random fashion. A user can take a license if it is not in use and the user holding the license is within communication range. The agents controlling the licenses restrict access to only group members who have departmental authentication (retrieved a priori), and are running on computers in the university domain. To retrieve a license, a user provides these three properties as credentials and attempts to perform an **in** operation for a license from a connected tuple space. If successful, the number of available licenses decreases by one. When the user finishes using the software, the agent replaces the license in its local tuple space.

To define the credentials in this application, an agent requesting a license uses the following code:

```
Credentials c = new Credentials();
c.selectProperty(HOSTPROFILE, ‘‘University’’);
c.selectProperty(HOSTPROFILE, ‘‘Department’’);
c.selectProperty(HOSTPROFILE, ‘‘Group’’);
```

The agent creates an empty credentials object and then selects three properties from the host's profile to add to the credentials: the university, the department, and the group. These three characteristics will be used to determine whether the agent has the right to access the license it requests.

Agents responsible for the licenses protect them by using access control functions that restrict access based on the administrative domains outlined above. This access control function is defined using the following code:

```
AccessControlPolicy policy = new AccessControlPolicy();
policy.addConstraint(‘‘University’’, ‘‘=’’, ‘‘WUSTL’’)
  .addConstraint(‘‘Department’’, ‘‘=’’, ‘‘CSE’’)
  .addConstraint(‘‘Group’’, ‘‘=’’, ‘‘mobi’’);
policy.addPermittedOperation(Operation.SINGLES);
acf.addPolicy(policy);
```

After creating a new policy, the agent adds three constraints to it that restrict the university, department, and group to the correct set of users. The agent permits all single operations (that interact with only a single license at a time). Finally, the policy is added to the access control function.

As these two examples demonstrate, the developer burden for adding access control to the application is minimized and builds on the notion of tuples and tuple spaces to ease the learning curve for the application programmer.

6 Discussion

The access control function provides a flexible mechanism for specification of dynamic and adaptive privileges in mobile systems. Next, we take a deeper look at two aspects of the access control function: expressiveness and overhead.

6.1 Expressiveness

While its expressiveness makes the access control function flexible and useful in coordination among constantly changing mobile agents, this flexibility comes at some cost. On one hand, because credentials can encode arbitrary information about an agent, particular applications can adapt credentials to their needs. In addition, because the access control function takes a number of parameters, an agent can dynamically adjust its policies. However, while complex policies are possible, constructing the function (from the developer's perspective) can become difficult as policies become more complex. Fortunately, because the design employs the use of pattern matching, much of this complexity can be hidden by the infrastructure.

6.2 Overhead

The addition of the access control mechanism introduces some amount of programming overhead, but this overhead is difficult to quantify without a case study involving users implementing actual access control policies. While this is a useful future task, it is outside the scope of this paper. Instead we focus on the overhead due to the additional communication and computation needed to provide the access control function described previously.

Additional Communication. The key aspect of the communication overhead is the amount of data (in bits) that must be sent. Before adding the access control mechanism, the number of bits required to send an operation request is: $b = |op| + |pattern| + |agent_id_r|$, where $|op|$ is the number of bits required to identify the operation; $|pattern|$ is the number of bits required to represent the pattern, which depends on the number of fields in the pattern; $|agent_id|$ is the number of bits required to identify the requesting agent so the response can be returned. It is likely that the pattern, which encodes the content-based nature of the request, dominates this expression, as the op and $agent_id_r$ are simple data types with small, constant lengths.

We can write a similar term to express the number of bits needed to be sent when using the access control function. This includes only the addition of the

number of bits necessary to encode the credentials: $b_{acf} = |op| + |pattern| + |agent_id_r| + |credentials_r|$.

Credentials are a tuple. Because tuples are similar to patterns, the number of bits required to represent the credentials is likely near the number of bits needed to represent a pattern. If so, the overhead of using access control is approximately 2. An application can directly control the amount of overhead it incurs because it determines what credentials to send with each request. In this respect, the use of application intuition to reduce the credentials transmitted to exactly those required reduces the communication overhead.

Additional Computation. Because the function can contain arbitrary code, its computational overhead lies in the hands of the application programmer. From the programmer's perspective, the operating conditions of the application must be a primary concern. If so desired, a system can include a mechanism to prevent undesirable access control functions by bounding the time they are allowed to run or by imposing restrictions on their capabilities. In most cases, however, the additional computation required is minimal since the access function may be limited to a pattern matching function.

7 Related Work

As discussed previously, the use of an access matrix does not directly lend itself to mobile systems. In one example of attempting to apply such a method, TuCSoN agents [18] are assigned capabilities defining tuple space operations for particular patterns in a certain tuple space. An access control list for the tuple space stores these capabilities. This approach requires that all coordinating parties are known in advance and that a centralized party can determine access policies statically.

Other systems use encryption for access control. In SecOS [19],, tuples are unordered sequences of individually encrypted fields, and, to match an encrypted field, a pattern must contain a correct key. Other work [20] associates keys with tuple spaces, and an agent must provide the key to access the tuple space. While both of these models provide access control mechanisms, they require secure key distribution and management, which affects the scalability of the system.

Law Governed Interaction (LGI) [21] provides an expressive approach to access control in which agents must adhere to a law that imposes context-sensitive constraints on the execution of tuple space operations. A law dictates actions an agent performs in response to tuple space operations. Programming applications in LGI requires programming specific actions in the access control policy and adding a controller to mediate tuple space requests. In contrast, in our model, programming takes place in the coordination model, and the agent's requested operation is checked with the access control function. One aspect of LGI that separates it from the access control mechanism described in this paper is that it allows access rules to be imposed from outside the individual agents. We do not consider such cases in our work because it departs from our view that agents should be as autonomous as possible.

The Smart Messages system [22] structures a mobile computing system in much the same way as discussed in this paper. Using Smart Messages, however, the coordination in the system occurs through the logical migration of Smart Messages. In this system, access control takes the form of admission control in determining when to allow migrating Smart Messages to execute on a new host. The admission managers responsible for this task use information about the resource needs of an arriving Smart Message as they relate to the available resources on the node. The access control mechanism described in this paper can account for more varied information than resource availability by using credentials describing the application agent and using the data items themselves when making access decisions.

Work targeted directly to ad hoc networks [23] begins to address the need for credential verification among interacting parties using X.509 certificates. This work focuses on adapting the chain of verification for certificates to function in an ad hoc network by using assertions generated by peers in the ad hoc network. The disadvantage of applying this type of solution in the environments we have described is that it requires some a priori knowledge shared among the peers in the ad hoc network in order to be able to verify the credentials of other participants. Key pre-distribution schemes targeted to sensor networks [24] have worked without a centralized server to establish pairwise secure communications. These approaches generally focus on maximizing the total security of the system to successfully handle more "compromised" nodes. These schemes focus simply on providing the ability to encrypt data and do not address the need to restrict access to certain data items based on contextual properties.

Additional work on authentication protocols in ad hoc networks [25, 26] focuses on securing communications among parties in ad hoc networks. These protocols tend to attempt to validate the identity of a communicating party. Our work instead focuses on the data sharing aspects and assumes that agents do not necessarily care about the exact identity of a coordinating partner, but about properties of the partner. This style of access control is more in line with our target environment since we assume that an application does not have a priori knowledge of the other agents or data it will interact with. The flexible nature of the access control mechanism described in this paper allows agents to base access decisions on abstract properties and the content of data, enabling more expressive access rules.

8 Conclusion

In today's emerging mobile systems, applications find themselves structured as networks of mobile agents that must interact to achieve the users' goals. As mobile devices become increasingly prevalent and more users join mobile networks, the complexity of mediating interactions among agents multiplies. A significant roadblock to the widespread deployment of many mobile applications lies in the inability to secure interactions in this open environment where encounters with others are necessarily opportunistic and unpredictable. The work presented

in this paper examined one aspect of this need by introducing a mechanism for agents to control access to data. This mechanism, in the form of an agent-tunable function, allows autonomously operating agents to share data with other connected agents, given some restrictions. Each agent makes individual access decisions for the data item it "owns" based on numerous properties including properties of the environment, of the agent's state, of the requesting agent, and even properties of the data item itself. By placing control in the hands of individual agents, we have eliminated the need for a centralized authority to make access decisions and thus created an access mechanism that functions in ad hoc networks where the coordinating parties are not known in advance. Because each access control decision is independent and made in a decentralized manner, the access control function naturally scales to networks of high numbers of mobile agents. Because we started with a foundational model of coordination, the resulting mechanism addresses the access control needs within mobile coordination models. In particular, the construct provides increased scalability and decoupling when compared with previous constructs without sacrificing flexibility and expressiveness.

Acknowledgements

This research was supported in part by the Office of Naval Research under ONR MURI research contract N00014-02-1-0715. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the Office of Naval Research.

References

1. Gelernter, D.: Generative communication in Linda. ACM Transactions on Programming Languages and Systems **7** (1985) 80–112
2. Kaufman, J., Lehman, T.: OptimalGrid: The almaden SmartGrid project: Autonomous optimization of distributed computing on the grid. IEEE Task Force on Cluster Computing **4** (2003)
3. Wyckoff, P., McLaughry, S., Lehman, T., Ford, D.: TSpaces. IBM Systems Journal **37** (1998)
4. Freeman, E., Hupfer, S., Arnold, K.: JavaSpaces Principles, Patterns, and Practice. Addison-Wesley (1999)
5. Murphy, A.L., Picco, G.P., Roman, G.C.: Lime: A middleware for physical and logical mobility. In: Proceedings of the 21st International Conference on Distributed Computing Systems. (2001) 524–533
6. Julien, C., Roman, G.C.: Egocentric context-aware programming in ad hoc mobile environments. In: Proceedings of the 10th International Symposium on the Foundations of Software Engineering. (2002)
7. Cabri, G., Leonardi, L., Zambonelli, F.: MARS: A programmable coordination architecture for mobile agents. Internet Computing **4** (2000) 26–35
8. Moore, J.: Mobile code security techniques. Technical Report MIS-CIS-98-28, University of Pennsylvania (1998)

9. White, J.: Telescript technology: The foundation for the electronic marketplace. General Magic White Paper, General Magic, Inc. (1994)
10. Gray, R., Kotz, D., Cybenko, G., Rus, D.: D'Agents: Security in a multiple-language, mobile-agent system. In Vigna, G., ed.: Mobile Agents and Security. Volume 1419 of LNCS. Springer-Verlag (1998) 154–187
11. Gray, R.: Agent tcl: A flexible and secure mobile-agent system. In: Proceedings of the 4th Annual Tcl/Tk Workshop. (1996)
12. Necula, G.: Proof-carrying code. In: Proceedings of the Symposium on Principles of Programming Languages. (1997)
13. Sander, T., Tschudin, C.: Protecting mobile agents against malicious hosts. In Vigna, G., ed.: Mobile Agents and Security. Volume 1419 of Lecture Notes in Computer Science. Springer-Verlag (1998) 44–60
14. Borselius, N., Mitchell, C.J., Wilson, A.: Undetachable threshold signatures. In: Cryptography and Coding—Proceedings of the 8th IMA International Conferemce. Volume 2360 of LNCS. (2001) 239–244
15. Papadimitratos, P., Haas, Z.: Secure data transmission in mobile ad hoc networks. In: Proceedings of the 2003 ACM Workshop on Wireless Security. (2003) 41–50
16. National Center for Supercomputing Applications, Integrated Decision Technologies Group: SAMCat: A securable active metadata catalogue. (2002)
17. Byrd, G., Gong, F., Sargor, C., Smith, T.: Yalta: A secure collaborative space for dynamic coalitions. In: IEEE 2nd SMC Information Assurance Workshop. (2001)
18. Cremonini, M., Omicini, A., Zambonelli, F.: Coordination and access control in open distributed agent systems: the TuCSoN approach. In Porto, A., Roman, G.C., eds.: Coordination Languages and Models. Volume 1906 of LNCS., Springer-Verlag (2000) 99–114
19. Bryce, C., Oriol, M., Vitek, J.: A coordination model for agents based on secure spaces. In Ciancarini, P., Wolf, A., eds.: Proceedings of the 3rd International Conference on Coordination Models and Languages, Springer-Verlag (1999) 4–20
20. Handorean, R., Roman, G.C.: Secure servise provision in ad hoc networks. In: Proceedings of the 1st International Conference on Service Oriented Computing. (2003)
21. Minsky, N., Minsky, Y., Ungureanu, V.: Safe tuplespace-based coordination in multi agent systems. Journal of Applied Artificial Intelligence **15** (2001)
22. Kang, P., Borcea, C., Xu, G., Saxena, A., Kremer, U., Iftode, L.: Smart messages: A distributed computing platform for networks of embedded systems. The Computer Journal Special Issue on Mobile and Pervasive Computing ((to appear))
23. Keoh, S.L., Lupu, E.: Towards flexible credential verification in mobile ad hoc networks. In: Proceedings of the ACM Workshop on Principles of Mobile Computing. (2002) 58–65
24. Du, W., Deng, J., Han, Y.S., Varshney, P.K.: A pairwise key pre-distribution scheme for wireless sensor networks. In: Proceedings of the 10th ACM Conference on Computer and Communication Security. (2003) 42–51
25. Weimerskirch, A., Thonet, G.: A distributed light-weight authentication model for ad hoc networks. In: Proceedings of the 4th International Conference on Information Security and Cryptology. (2001) 341–354
26. Balfanz, D., Smetters, D.K., Stewart, P., Wong, H.C.: Talking to strangers: Authentication in ad hoc wireless networks. In: Network and Distributed System Security Symposium. (2002)

Separation of Concerns for Mechatronic Multi-agent Systems Through Dynamic Communities*

Florian Klein** and Holger Giese

Software Engineering Group, University of Paderborn,
Warburger Str. 100, D-33098 Paderborn, Germany
{fklein,hg}@upb.de

Abstract. Multi-agent systems present a promising paradigm for coping with the complexity of intelligent mechatronic applications, particularly where purposeful behavior and complex structures emerge from the interactions of seemingly simple elements. The safety of mechatronic systems relies on predictability, which is apparently at odds with the concept of emergent behavior. When designing complex mechatronic multi-agent systems, the main challenge thus lies in achieving predictability without ruling out the desired emergent behavior. We propose to achieve this by decomposing the requirements and design into largely independent concerns, represented by social structures with behavioral norms, which are reconciled at the agent level. An explicit grounding of all constructs in observable entities from the mechatronic system's environment model makes them amenable to formal analysis and enables rapid prototyping.

1 Introduction

The field of mechatronics [1] combines mechanical, control, electrical, and software engineering. Its aim is to improve the performance of mechanical systems by embedding them with intelligent electronic controllers that analyze and exchange information and adapt and coordinate behavior. Due to their relative autonomy, these controllers are typically interpreted as agents. Applying the multi-agent system paradigm to describe their interactions allows the mechatronics community to build on a wealth of experience concerning the design of distributed information systems.

The RailCab project[1] is a prominent effort to develop such a mechatronic multi-agent system. Fleets of intelligent shuttles, capable of transporting a small number of passengers or a cargo container, autonomously navigate a passive track system and make independent and decentralized operational decisions. The underlying vision is to combine flexible, on-demand scheduling with extreme cost and resource effectiveness, thus combining the specific advantages of individual and public transportation. Even though shuttles compete by bidding on transportation tasks, they may collaborate in

* This work was developed in the course of the Special Research Initiative 614 - Self-optimizing Concepts and Structures in Mechanical Engineering - University of Paderborn, and was published on its behalf and funded by the Deutsche Forschungsgemeinschaft.

** Supported by the International Graduate School of Dynamic Intelligent Systems.

[1] http://www-nbp.upb.de/en

R. Choren et al. (Eds.): SELMAS 2004, LNCS 3390, pp. 272–289, 2005.
© Springer-Verlag Berlin Heidelberg 2005

energy-efficient ad-hoc convoys. Their most advanced aspect is their ability to reevaluate their goals using a novel process termed self-optimization, which allows them to evolve their strategies and behaviors in response to their environment.

From the software engineering point of view, the key challenge is constraining this process to safe behaviors. Mechatronic systems are normally safety-critical, as failures in the control software affect a mechanical system that might potentially harm people or the environment. The classical approach to mechatronic system design therefore strictly relies on analytic predictions of a system's properties in order to guarantee that a design meets a set of carefully elaborated safety requirements. Now, self-optimization necessarily introduces a certain amount of unpredictability into the system.

Networking self-optimizing systems multiplies this effect. Unfortunately, many mechatronic solutions specifically derive their power from the cooperation between agents. A particularly fascinating effect in this respect is the emergence of complex behaviors or structures from simple, local interactions.

This is the focus of the subfield of swarm intelligence [2]: How can a set of simple rules makes a multitude of individual agents behave in a coordinated fashion? When designing such systems, the main challenge lies in managing the inherent combinatorial complexity and handling the way individual agents mutually affect each other. Successful agent-based solutions are often inspired by analogies to natural or social phenomena, not least because developers find such metaphors a useful or even essential tool for understanding complex systems. Designing or even understanding dynamic multi-agent systems is usually not possible in a purely analytical fashion.

While such inaccessibility to analysis and a safety-critical system's need for analytic predictions seem to be diametrically opposed, we hope to be able to reconcile both approaches. In order to enable the use of self-optimization and emergence in mechatronic systems, we propose to separate different system concerns at the multi-agent system level by assigning them to specific agent communities. Conflicts between the prescriptions of different communities are then resolved locally in each agent. The system designer maps each requirement to a particular system concern, which specifically allows the extraction of safety-related requirements. As each concern can, to some extent, be studied in isolation, this results in a reduced complexity of the overall design. Possible conflicts at the intersection points between concerns can be systematically identified and resolved using a combination of formal methods and experimental validation.

Both techniques are enabled by the principle of explicit grounding: All abstract concepts such as communities or promises need to be expressed in terms of observable entities from the mechatronic system's environment model. This provides a shared vocabulary for the formal specification of the concerns' required properties and a common base for merging different concerns, which is necessary for conducting a formal analysis of their composition. It equally provides operational semantics for the abstract concepts, which facilitates the execution and thus early testing of models. Appropriate tool support would allow a prototyping approach with rapid cycles of experimental evaluation and subsequent refinement of the design specification. However, explicit grounding restricts the types of agent behavior we can express to an executable subset of standard software engineering concepts, which in turn limits our ability to model sophisticated cognitive capabilities.

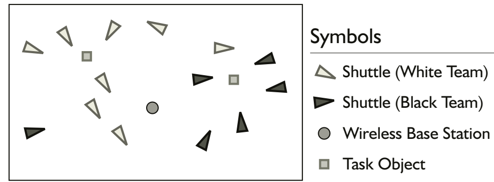


Fig. 1. Elements of the physical model.

Our approach is nonetheless sufficient for reasoning about the reactive and proactive behaviors of mechatronic agents and their motivations. We illustrate this with the help of an example inspired by the RailCab project, which we are going to use throughout this paper.

In the following section, we introduce the physical domain model. It serves as the basis for the conceptual models in Section 3. Different concerns of the system are defined by modeling and decomposing the requirements (Section 3.1), and assigned to specific responsible communities that are fleshed out in some detail (Section 3.2). In Section 4, the composition of the concerns is discussed, dealing with local conflicts (4.1) and conflicting commitments (4.2). The resulting specification is then used for rapid prototyping and exploration in Section 5. We first discuss its operationalization (Section 5.1) and then execute it to identify and resolve undesired emergent behaviors (Section 5.2). We conclude with a short summary and an outline of our future plans.

2 Physical Domain Model

As it is directed at enabling the exploration of ideas and strategies through rapid prototyping [3], our treatment of agent-related high-level concepts closely builds on established software engineering practices. The physical domain model, which is basically the system's fundamental ontology expressed as a UML class diagram, describes the concrete entities that are present in the system, i.e. consists of the agents themselves and a model of the perceivable environment as it presents itself to their sensors. It serves as an explicit conceptual and technical grounding for all the more abstract concepts we subsequently introduce. In this way, we hope to benefit from both the power and expressiveness of abstract reasoning and the pragmatic elegance of non-symbolic intelligence and physical grounding [4].

As we are discussing mechatronic systems, physical reality offers itself as the evident choice as the object of a - literally - physical domain model. Specifying such a model already is a common design activity, as control software is usually designed and tested within a simulated environment that faithfully models the actual sensor input used in later development phases and the production system. As the elements of the model refer to physical objects and their perceivable attributes, their semantics are quite straightforward and intuitive.

In order to focus on the exploration of our methodological ideas, we used a simplified domain model based on the RailCab application as a test case for the application of the approach we present in this paper. As structuring the coordination and control problems inherent in the application by means of separate concerns is the focus of our

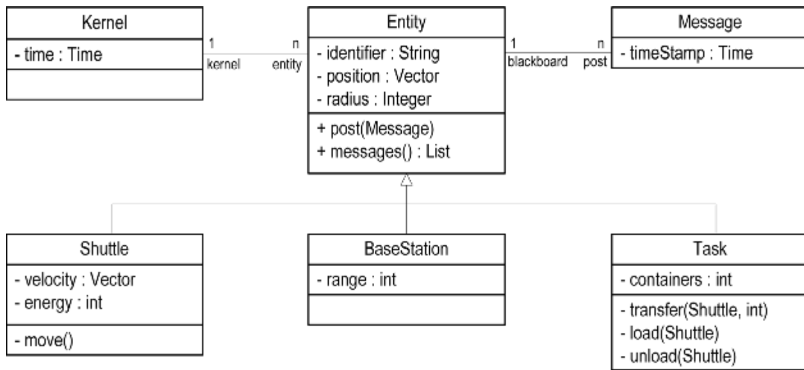


Fig. 2. The physical domain model.

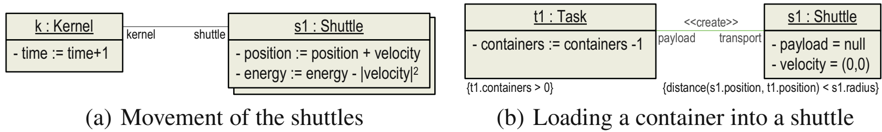
approach, we kept the envisioned coordination architecture largely intact, but allow the shuttles to move freely across a plane (see Figure 1). The use of tracks would make distances between shuttles depend on their current tracks (discrete) and their relative positions on those tracks (continuous), which would needlessly encumber several diagrams with case differentiations without adding anything on the conceptual level.

In our example, the physical model mainly consists of physical entities, i.e. the shuttles, transportation tasks manifested as stacks of containers, and wireless base stations. A kernel provides a simulated plane and a discrete system-wide clock. Each entity may receive and publish messages, providing a simple blackboard-type messaging service.[2] Entities have an identifier and positions, shuttles have a velocity and an energy level, tasks have a certain number of containers left, and base stations have a maximum transmission range (see Figure 2).

This static model may now be augmented by a specification of its basic dynamics. This physical process model is once again restricted to the immediate semantics of observable 'physical' actions. Obviously, this rules out the description of more complex behaviors, cognitive processes and deliberated interactions. The model is, however, quite sufficient to describe the effectors that agents use for the manipulation of their environment. It thus provides a purely 'phenomenological' description of the behaviors exhibited by the system, which is all that is required at this point.

As a formalism to describe these processes, we use story patterns [5], which will keep playing a prominent role throughout the whole development process. Story patterns are an extended type of UML collaboration diagram based on the theory of graph grammars [6]. Their appeal is in their ability to formally express both constraints in the vein of the UML's Object Constraint Language (OCL) and behaviors, described as the transition between two instance situations serving as pre- and postconditions, with one straightforward graphical notation. Figure 3 shows two simple examples. Patterns may impose constraints both on structure (one kernel, one shuttle) and attributes (pay-

[2] Though we abstract from the messaging infrastructure, neither our approach nor the application example are principally limited to centralized messaging. A distributed, decentralized solution is entirely feasible, but would introduce replication issues that would distract from the paper's focus.



(a) Movement of the shuttles (b) Loading a container into a shuttle

Fig. 3. Definition of basic dynamics.

load=null). If these constraints are successfully matched against the instance graph at run-time, the pattern is executed and may change the graph structure (create or delete objects and associations), modify attributes (time:=time+1) or trigger methods invocations.

Figure 3(a) shows the process for shuttle movement. When moving, shuttles expend energy at a rate proportional to the square of their speed, $|velocity|^2$. Shuttles may form and dissolve energy-efficient convoys, which affects the rate. For tasks, we define the following processes: creation, loading containers into and unloading them from shuttles, paying shuttles by transferring energy, and termination. Figure 3(b) displays the story pattern for loading a container. Furthermore, all agents publish messages. Even though the scope of possible actions may seem extremely limited at this level, it serves as the setting of rich and complex interactions taking place at a higher level of abstraction, including negotiations and cooperative problem solving.

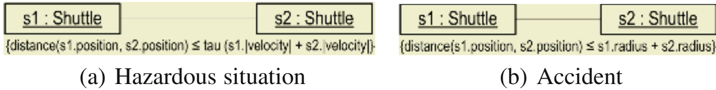
3 Conceptual Model

These higher levels of abstraction can now be described by building on the semantics of the physical domain model. Increasingly complex definitions may be used as building blocks in further definitions describing goals, interaction protocols or basic deliberations. In our simplified example, this is a relatively direct process leading straight from the physical model via requirements to structures and behaviors. For more complex systems, a more intricate iterative process going back and forth between the different concerns of the model is usually necessary.

3.1 Requirements

The physical domain model now provides us with a basic vocabulary to state requirements. For the needs of our example application, the ability to describe 'hard' goals that can be precisely defined in terms of the physical model is sufficient. They can easily be expressed using story patterns. The primary goal of the shuttle agents is to maximize their energy level, which can be decomposed into maximizing the number of tasks completed and minimizing the energy consumption. Tasks strive to reach their destination both as quickly and as cheaply as possible. These subgoals are contrary to each other and therefore need to be adequately balanced. A secondary but nonetheless very important concern is safety: a collision between two shuttles needs to be avoided at all cost. The hazard and an actual accident are formalized by Figures 4(a) and 4(b).

Due to interdependencies, the progression from the physical model to the requirements will usually not be as strictly sequential as it is presented here: requirements may motivate the introduction of new entities and possible actions, which may in turn lead



(a) Hazardous situation (b) Accident

Fig. 4. Collision between two shuttles.

to additional requirements. This can be seen even in this simple example: The ability to form convoys, itself clearly motivated by the goal of minimizing the energy consumption, entails the introduction of new subgoals and rules governing convoys.

When attempting to build verifiably safe multi-agent systems, reconciling a set of possibly conflicting requirements is a central conceptual problem. Especially large-scale multi-agent systems can quickly lead to intractably complex designs with a quickly exploding number of possible states. Considering the usually limited resources of embedded processors, computation may also pose problem at runtime.

We aim to reduce the complexity of the requirements by decomposing them into largely independent concerns along different, ideally orthogonal, axes. Hierarchical, spatial or temporal separation criteria are most commonly used. Each of the resulting subsets is then assigned to the concerned group of agents, who form a community charged with its implementation. If we can minimize the dependencies between different communities, it becomes much easier to prove that the behaviors they stipulate are indeed compatible and composable in a safe manner. This will then allow agents a certain freedom to dynamically join and leave communities, as long as they stay within the bounds established by the formal analysis. A clean separation between safety-critical and efficiency-related concerns further allows the use of the appropriate design and verification tools and methodologies for each problem.

In the example, there are communities responsible for the publication and for the auctioning and execution of tasks. The safety concerns are addressed by smaller, physically localized communities that form around the wireless base stations. Finally, the shuttles team up in virtual companies that coordinate the members' distribution and bidding strategies in order to maximize their profits. Shuttles working for the same company may spontaneously form convoys, which are organized as temporary ad-hoc communities.

3.2 Communities and Cultures

Communities and Cultures are the cornerstones of our approach's conceptual model. They structure and connect many other high level constructs of the model. Cultures are abstract sets of norms, which include concepts as diverse as roles and behaviors, message formats and social conventions concerning the interpretation of behavior. Communities are concrete groups of agents that implement a particular culture.

A community may either exist as an object in its own right, like a company or a mailing list, or just implicitly defined by its members, like a convoy or the community of all shuttles near a particular base station. Likewise, membership in a community may be established explicitly through some attribute or relationship or implicitly through performing a specified behavior. Either way, membership in a community is a directly observable property on the physical level, be it as an entry on a membership list (in

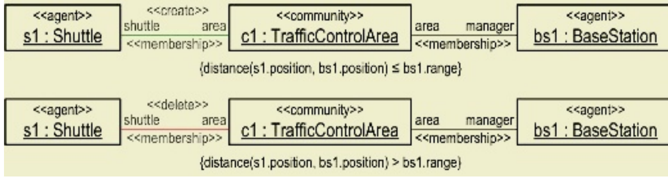


Fig. 5. Implicitly defined membership in a Traffic Control Area.

the form of a published message) or as a shuttle's current position. This holds true for all subsequently defined constructs: they are all associated with concrete, though possibly complex, observable patterns in the physical model. A community is therefore associated with two story patterns describing the conditions for joining and leaving it. Figure 5 provides the corresponding patterns for the physically localized traffic control communities. In the diagrams, cultures, communities and agents are marked with corresponding UML stereotypes that serve as a compact notation for classifying objects with respect to these conceptual categories. Likewise, membership, the special defining relationship that exists between a community and its agents, is qualified using a stereotype.

A culture specifies the different roles that occur in a community, complete with constraints that permit or require their adoption. A role in turn specifies a collection of behaviors with rules for their invocation, expressed by story patterns. The behaviors themselves are described using statecharts with real-time annotations.

Behaviors are closely linked to social conventions, which are an abstract but nonetheless influential part of a culture definition. Social conventions describe the rules that apply to a community. Normative rules usually prescribe a certain operative behavior, e.g. which shuttle should have the right of way when they are on a collision course. But in order to model more complex social interactions, it is also useful to have descriptive social conventions that offer a definitive interpretation of an agent's behavior. In this context, a concept derived from speech act theory plays an important part: 'Professed intentions' publicize an agent's intentions and may be classified using speech act classes such as assertion, permission, prohibition, directive or commitment [7]. A professed intention can be issued explicitly, usually as a message that has been defined by an interaction pattern or that conforms to a more generic shared communication language, or implicitly, basically as an interpretation of certain acts that has been agreed upon beforehand. Even though professed intentions are essentially abstract, they are always tied to an unambiguous manifestation in the physical model and can therefore be modeled by story patterns. In the models we present, the manifestation of a professed intention is marked with the respective stereotype. The actual content of the professed intention is specified independently from the manifestation by means of a dedicated object, which is associated with the originating agent using the *intention* stereotype. The object contains a pattern that specifies e.g. what is asserted or promised. As professed intentions can reference other professed intentions, these patterns can serve to introduce complex behaviors into a system. E.g. once a commitment to complete a task is fulfilled, the same pattern that assesses this might trigger a new commitment that obliges the client to pay.

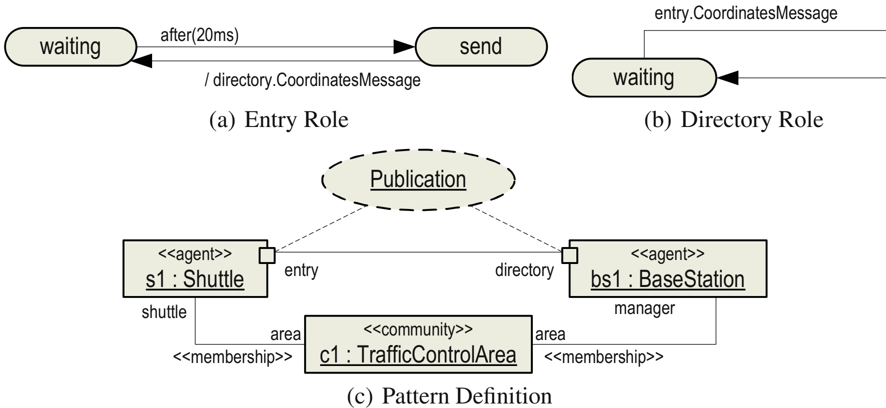


Fig. 6. Shuttle Information Publication Pattern.

3.3 Modeling Concerns

We will now present some detailed examples from our specification. Putting safety first, we start out with the local traffic control communities responsible for avoiding collisions. It is generally recommended to design the safety-critical concerns of a system first and submit them to formal verification. They can then serve as a safe base for further experiments concerning emergent behavior and advanced optimization strategies.

The rules for establishing the membership in a traffic control community have already been given in Figure 5. The second rule stipulates that upon joining the community, a shuttle immediately start executing the publication pattern described in [8] and illustrated by Figure 6(c): Every 20 time steps, each shuttle sends its position and velocity to the respective base station (see Figure 6(a)), which receives (see Figure 6(b) and publishes it, and reads the currently available information about the other shuttles. Failures of any kind trigger an emergency stop. The actual collision avoidance pattern is defined building on this infrastructure: Shuttles generally head towards their targets at their desired cruising speed. On the station's list, they check for shuttles whose time to (potential) collision falls below a certain limit at the current velocities. Shuttles heading toward them exert a repulsive force that grows as their tau, the distance remaining over closing speed, decreases. The sum of all forces determines the new velocity and thus initiates the necessary evasive action. Note that this includes the option to stop altogether, the trivially safe choice.

This pattern only works for shuttles in the same traffic control community that can perceive each other via its blackboard. The areas covered by different local communities overlap in order to avoid collisions upon joining: the layout ensures that the agents are already acquainted in at least one of the overlapping communities (see Figure 7).

We now turn to the actual purpose of the system, the completion of tasks. As this concern is not in itself safety-critical, but requires a greater degree of flexibility due to its complex and more abstract nature, we do not use statecharts, but story-pattern-based rules to specify its behavioral patterns. A task starts an auction by publishing an *AdvertiseTask* message, thus creating and joining a dedicated community (see Figure

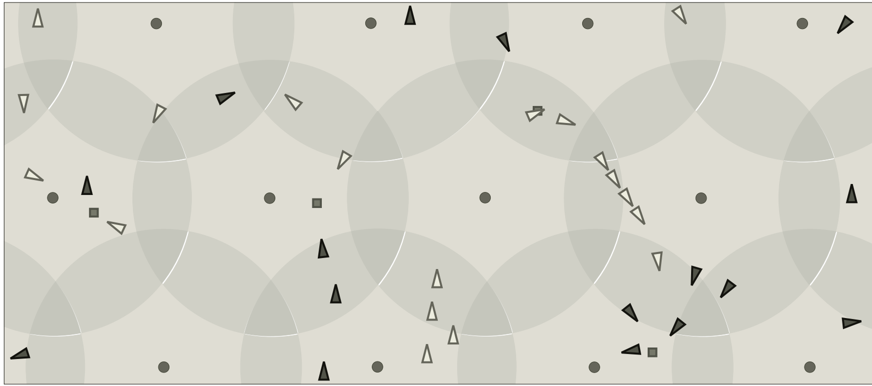


Fig. 7. Base stations with associated areas.

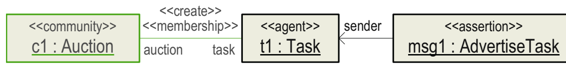


Fig. 8. A task initiates an auction.

8). The culture of these auction communities governs all the economic aspects of the system. The AdvertiseTask message contains the intended destination. As it publishes information about the task's internal state, it is an assertion.

If a shuttle is interested in the transportation task, it posts a *PlaceBid* message containing a bid (expressed as energy, which is used as currency) and a proposed deadline for the completion of the task to the auction community's message list. This message both implicitly establishes the shuttle's membership in the task's auction community and represents a commitment to honor the offer it proposes. The actual intention it professes is given by the story pattern in the *WorkPlanCommitment*: delivering the task to its destination before the promised deadline (see Figure 9).

A task will keep accepting bids for its auction as long as there are containers left. It constantly ranks all proposals according to its preferences with respect to a low price or an early time of arrival. As a task needs a number of shuttles equal to the remaining number of containers, it will always accept the top $containers$ bidders. Thus, the publication of a *RateBids* message implies a concrete commitment to load a container into any top ranked shuttle that presents itself at the task's position (see Figure 10).

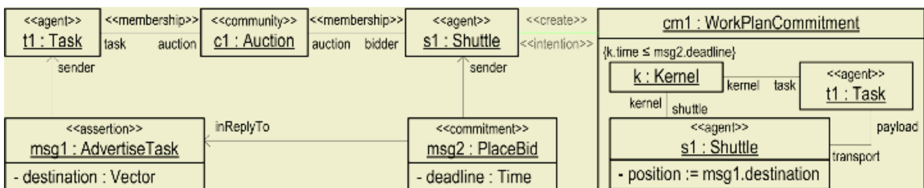


Fig. 9. A shuttle places a binding bid.

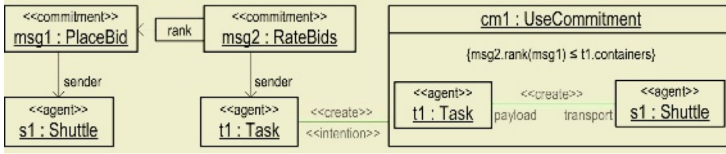


Fig. 10. A task ranks all incoming bids.

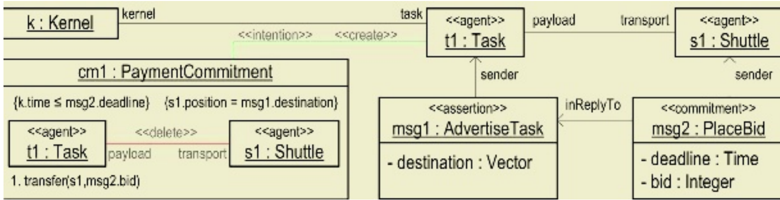


Fig. 11. Tasks commit to paying once a shuttle has kept its promise.

The act of loading a container into the shuttle entails another, rather important commitment: that is, to unload the container and transfer the promised energy as soon as the shuttle's contractual obligations have been fulfilled (see Figure 11).

In this manner, the culture provides a fairly high-level description of the interactions surrounding the assignment and execution of tasks that still offer precise operative semantics.

The last culture we present pertains to companies. Shuttles may form companies by posting *DeclareAffiliation* messages that assert their membership (see Figure 12).

Shuttles within the same company coordinate their behavior with respect to bidding and strategic movements in a peer to peer fashion. One possible coordination behavior open to a shuttle that is faced with a large, attractive task is asking for assistance by issuing a *directive*. This 'orders' any shuttle that cares to respond to it to a certain destination (see Figure 13). If, however, a shuttle replies favorably to the directive, this is once again a concrete commitment to fulfill a *WorkPlanCommitment* that needs to be honored.

There finally is a culture used by the ad-hoc convoys created by shuttles from the same company (as seen in Figure 7). Each convoy corresponds to a temporary community. While member of a convoy, all shuttles repeatedly issue short-term commitments to maintain a common velocity, which is what enables traveling in such close proximity without triggering the collision avoidance pattern and prompting evasive action.

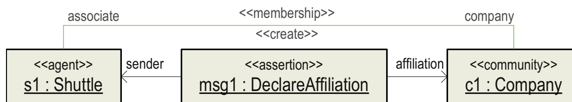


Fig. 12. Membership in a Company.

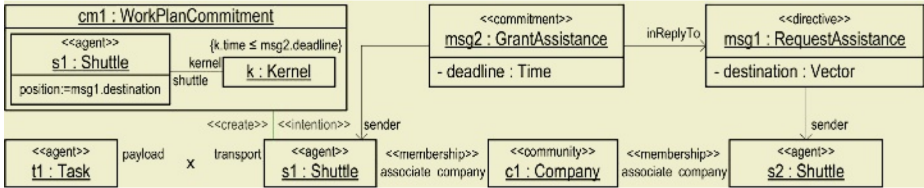


Fig. 13. Shuttle promising assistance.

4 Composition of Concerns

In order to integrate the separate concerns into a coherent whole, they are combined and reconciled locally in each agent. This step is not trivial, as the agents need to conform to the rules, requirements and constraints from each individual concern, without the benefit of an unambiguous specification concerning the way each rule is to be fulfilled or what the agents' internal design should be like. The conceptual specification merely implies certain very generic assumptions about the cognitive model of an agent. The physical model serves as an ontology that is to be shared by all agents. A certain amount of knowledge about the environment is indispensable for the evaluation of the rules governing the interaction patterns and generating the messages they require. In this context, the perspective on an agent's cognition is purely external: Through social conventions and behaviors, we define a model of what we suppose or require an agent to know in a specific situation. For example, we require an agent to know about the positions of all adjacent shuttles as published by the local community. This 'legal perspective', which is inspired by the way human law contains provisions that apply if a defendant 'knew or *had to know*' a fact, allows us to abstract from the agents' internal knowledge representation and even give a limited operational definition of such elusive concepts such as 'truthfulness'. As the analogy implies, this entails that a system either has to respect the conventions by design or make provisions for handling violations. The legal perspective also opens up a path to the simulation of incomplete designs: By non-deterministically choosing between the behavioral options offered by the specification, the behavior of agents whose logic has not been implemented yet can be approximated.

4.1 Safety-Critical Conflicts

The integration may consequently result in the most general possible agent specification that conforms with the legal specification, even though designing more specific, elaborated agents is entirely acceptable. When dealing with the safety-critical concerns of the system, a state-model needs to be constructed that does not violate any of the safety properties required of the system. This often challenging and time-consuming manual task might be facilitated or even eventually made unnecessary by automated construction methods and tools. An approach that allows the synthesis of composite behavioral models out of non-orthogonal concerns without real-time constraints is presented in [9]. Extending this work with respect to the statecharts with real-time annotations used for the specification of the design patterns seems like a viable option in this context.

Once an integrated specification of the safety-critical real-time concerns of the system has been created, its correctness with respect to the original concerns and the required safety properties can be formally verified. As we present compositional model checking, the approach used to achieve this, in detail in a dedicated paper [8], we shall only provide a brief summary in this context, though.

Design patterns [10] for component behavior define roles with well-defined required real-time behavior and communication channels. Certain safety and liveness properties that are supposed to hold for the mechatronic system in question are translated into temporal logic and subsequently locally verified for the isolated design pattern by means of a model checker. The design patterns are then composed in strict accordance with a set of compositional rules that syntactically only permit consistent component structures. By locally checking the parallel composition of the required behaviors within each component, conflicts resulting from the composition can easily be identified. In this context, first results for the automatic resolution of such conflicts for the untimed case have been developed [11]. At the cost of the restrictions introduced by the compositional rules, it is thus possible to avoid the state explosion problem and verify large systems using only a limited number of efficient, localized checks.

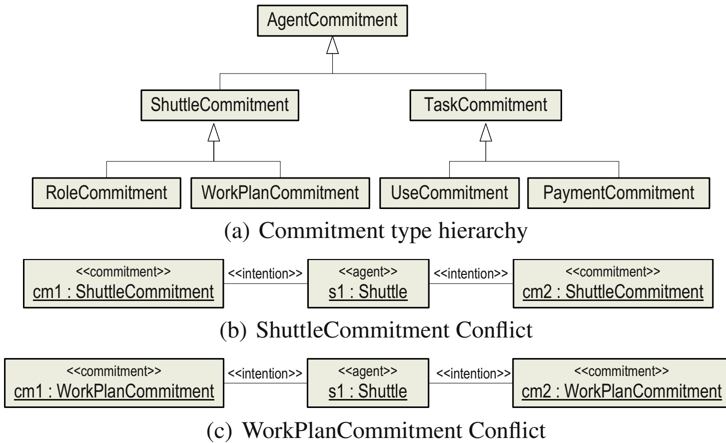
In the current approach, agents concurrently participating in several communities can be seen as analogous to components implementing several roles safely and consistently and may thus be verified by means of compositional model checking. In our example, we can verify the system's safety concerning collisions using this method.

4.2 Commitment Conflicts

As the rule-based specification of the performance-related concerns of the system involves more complex models and predetermines fewer behavioral details, automatic synthesis of an agent's behavioral model seems hardly possible in this context. As there are, often intentionally, many different behaviors that respect the rules, and the construction of an integrated behavioral model that reflects all those possibilities is generally not possible, we restrict ourselves to uncovering conflicts at the specification level at this point. Actually designing the agents' behavior is a manual process that requires appropriate decisions by the developer at a later time.

When checking for conflicts between different commitments, we can employ approaches for the verification of graphical specifications based on graph grammars [12] and their consistency [13]. We start by compiling the hierarchy of the different abstract categories of commitments found in the shuttle system (see Figure 14(a)). We then focus on the commitments made by a particular agent type, say shuttles. We check for the rule in Figure 14(b), i.e. whether a shuttle can issue two concurrent commitments. As shuttles issue different types of commitments, this is certainly the case. Closer inspection does not necessarily indicate conflicts, though, as the level of abstraction is too generic.

We therefore need to refine our analysis and specifically look at different types of shuttle commitments. Figure 14(c) proposes the rule that a shuttle should not issue two commitments concerning its work plan at the same time. As the deadlines are tightly calculated, fulfilling both commitments will only be possible when their respective des-



(a) Commitment type hierarchy

(b) ShuttleCommitment Conflict

(c) WorkPlanCommitment Conflict

Fig. 14. Analyzing commitments.

tinations chance to be in close proximity, i.e. the concurrent commitments generally represent a genuine conflict and should indeed be prohibited.

Analysis reveals that the specification allows the concurrent posting of a *PlaceBid* (see Figure 9) and a *GrantAssistance* (see Figure 13) message, which imply two conflicting *WorkPlanCommitments*. This conflict becomes possible because placing a bid already constitutes a commitment, whereas the second pattern only checks whether the shuttle is currently carrying a payload. This problem can easily be remedied by changing the rules so that a shuttle can only commit to a work plan if there is no preexisting commitment to another active work plan.

The structured composition between the safety-critical and the performance-related concerns ensures that the system's safety properties are not invalidated by the behaviors stipulated by rules and commitments. The result of this step is therefore a safe behavioral specification for the system's agents, which can be independently verified by re-running the model checker on the integrated model.

5 Rapid Prototyping

Now that we have modeled and integrated all relevant aspects of the agents' behavioral model, we can move on to the experimental evaluation of the system's emergent behavior. As the system's safety properties have been formally verified, the focus here is its efficiency. We first discuss issues related to the operationalization of the specification and then use it to detect and resolve an emergent pathological behavior.

5.1 Operationalization

The legal perspective primarily provides an 'interface specification'. With respect to an agent's cognition, it ultimately needs to be complemented by a more detailed design dealing with encodings, knowledge representation or complex inference mechanisms. For the rapid prototyping of the system's design, i.e. the rules and control structures,

this is neither practical nor actually desirable. A concrete implementation might introduce implicit limiting assumptions not warranted by the specification, whereas nondeterministic choice ensures the consideration of all admissible behaviors. Nonetheless, the agents' internal cognitive structures and processes may be supplied and elaborated in a process of stepwise refinement, which is useful for the validation of the completed parts of an incomplete implementation.

At the level of detail that the specification of the legal perspective provides, it is thus already possible to implement a prototype to experimentally test the design. Because of our emphasis on the design principle that every concept needs to be grounded in the physical domain model, the specification, including the declarative parts, has precise operational semantics and can directly be turned into an executable model. As code generation is available for both class diagrams and story patterns, this enables rapid development cycles of experimental evaluation und subsequent refinement of the design specification.

Fujaba[3] is an open source case tool that has been developed at the University of Paderborn and is currently being extended in cooperation with several other universities. It is capable of generating complete executable programs from specifications consisting of UML class diagrams, statecharts and story patterns. Currently, Java is the only supported target platform, but efforts to extend the tool to C++ are under way. A companion application to Fujaba is DOBS (Dynamic Object Browsing System), a visualization tool that dynamically generates interactive object diagrams at run-time using the Java reflection API. DOBS facilitates the visualization of structures and structural changes and doubles as a graphical debugger. It was used successfully for the rapid prototyping of production systems [14]. Using Fujaba, the specification outlined above can be modeled and exported to Java source code. In conjunction with a pre-built generic simulation framework that provides the domain independent aspects of the simulation, such as a basic messaging framework, object management and a threading model, this provides a basic simulation environment for control patterns in the RailCab domain.

5.2 Emergent Pathologies

We can now execute the specification we designed. The safety-critical concerns that have been formally verified should behave as predicted. The emergent behaviors, however, might not materialize or turn out in unexpected and undesired ways. Assuming that each concern is correctly implementing its own requirements, this is usually the consequence of interference between two or more concerns.

There is no general recipe for resolving conflicts between concerns, as balancing the different requirements usually depends on domain-specific knowledge. Consider the case when an expected emergent behavior is modified or inhibited by a safety-critical concern of the system. Even though safety clearly has the higher priority, only an in-depth analysis of the conflict can ultimately resolve it: If the safety-critical part of the system explicitly inhibits the desired behavior, the behavior is either inherently unsafe and should be dropped, or the constraints of the safety-critical part are too restrictive and should be relaxed. E.g. a rule forbidding shuttles to move would be quite safe, but

[3] http://www.fujaba.de

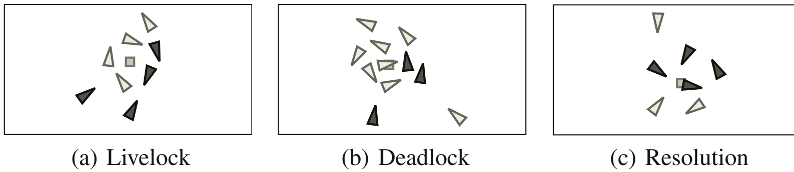

(a) Livelock (b) Deadlock (c) Resolution

Fig. 15. Resolution of pathological behavior during task pick-up through priorities.

make the system completely ineffective. If, however, the inhibiting behavior is optional, the affected concern should be augmented by valid refinements of the respective pattern that avoid this behavior. We provide an example of this in the following.

The control mechanisms we designed in the previous section generally work as intended. Most notably, they reliably avoid collisions. When introducing tasks requiring a large number of shuttles, this in turn leads to a long list of accepted bidders which will then simultaneously attempt to approach the task entity. Though collisions are avoided, a pathological behavior ensues: due to the collision avoidance pattern, the interested shuttles effectively keep each other from approaching the task and start – apparently aimlessly – moving around it in pulsating circles (see Figure 15(a)), a classic case of livelock. In a second pathological pattern, a single agent actually reaches the task and picks up a container, but is then blocked from leaving by the approaching other shuttles (see Figure 15(b)) in a deadlock situation.

By introducing a new behavioral pattern into the culture of the communities grouping the shuttles bidding for the same task, both problems can be remedied. According to their rank on the list of accepted bidders published on the community's blackboard, shuttles are assigned a priority. When two shuttles approaching the same task interfere with each other, the shuttle with the lower priority will exhibit a stronger evasive impulse than required by the collision avoidance pattern, i.e. it will stop more quickly, wait and even back away (see Figure 15(c)). The dominant shuttle is still bound by the collision avoidance pattern, but will not have to perform significant evasive maneuvers as the conflict is unilaterally resolved by the other shuttle. Conceptually, the new pattern is thus layered on top of the existing collision avoidance pattern, effectively eclipsing it but leaving it intact. As the new behavior it specifies is still entirely compatible with the basic pattern, the safety guarantees made by the collision avoidance pattern still hold.

6 Related Work

The principle of *separation of concerns* [15] has recently drawn a lot of attention due to advances like aspect-oriented programming (AOP) [16] or subject-oriented programming (SOP) [17]. The general idea is to consider different aspects or views on the system in isolation and only compose the overall system design or implementation at the end of the process. Thus, it becomes possible to focus the development effort on individual concerns and develop suitable local solutions, and at the same time facilitate an understanding of the complete system. Cross-cutting concerns like persistence, logging or error handling are frequently cited examples of aspects that are ideally suited for such treatment. In contrast, the presented approach addresses overlapping functional concerns and their systematic composition.

Aspect composition is usually carried out at the source code level, whereas our approach operates at the specification level. One notable exception is *subject-oriented design* [18], which also is specification-based. The approach uses individual *design subjects* to synthesize object-oriented design models, but is limited to concepts for the composition of their structural features. *Role-based modeling* [19] is another related approach that supports several dedicated views on a system or a component which are subsequently combined. In addition, a tool for weaving aspects described by means of UML role models with additional OCL constraints is sketched in [20]. It combines the idea of aspects at the design level with role modeling. However, the proposed weaving and superposition techniques only address the composition of structure and method bodies, while the presented approach focuses on the behavioral composition of the reactive behavior.

Current proposals for a separation of concerns for multi-agent systems [21] largely focus on exploiting separation of concerns within the individual agents to ease their implementation. The presented approach is different in using social interaction rules (cultures) and their structuring (communities) to separate different aspects of the overall design of a multi-agent system.

With respect to our approach to social structure in multi-agent systems, we see strong parallels to the current work on organization centered multi-agent systems (OC-MAS) based on the agent, group, role (AGR) model [22]. Common points include the predominance of inter-agent aspects and the abstraction from agents' cognitive abilities. However, we apply dynamic, intersecting groups as a more general, implementation-agnostic modeling concept.

Rapid prototyping is a method in wide-spread use in many different areas. In the specific context of mechatronic systems, it is often concerned with the design and incremental improvement of control laws (cf. [23]). For this purpose, it is combined with virtual prototyping (cf. [24]) or, more frequently, dedicated prototyping hardware (e.g. FPGAs) that allows a quick implementation and reconfiguration. Here, the control structures and dataflow are usually rather static, however. Software engineering, especially with proper CASE tool support, lends itself to rapid prototyping (cf. [3]). While it is generally seen as a useful method for early validation, as recently popularized by approaches like the test-driven Extreme Programming [25], it is frequently not combined with a formal process. Where applied systematically, it is often primarily seen as a tool for requirements engineering [26]. It has also been advocated that prototyping is an appropriate approach to support aspect composition (cf. [27]). In contrast, the presented approach uses model-based prototyping to identify potential interactions between agents that need to be coordinated and to explore and refine emergent behavior, while conformance of multiple composed concerns within each single agent is addressed by formal verification techniques.

7 Conclusion and Future Work

We have presented an approach for the design of mechatronic multi-agent systems which addresses the demand for the integration of partially predictable and partially emergent behavior. We address the complexity problem by decomposing the system

into concerns that, to a great extent, allow the requirements and design to be studied independently. The concerns are realized using social structures (communities) with behavioral and communicative norms (cultures). The concluding composition of the concerns is effected in a manner that preserves all required analytic properties but lets complex emergent behavior further refine them. Conflicts are systematically identified and resolved at different levels: safety properties and the consistency of commitments are verified locally for individual agents, whereas emergent pathologies are identified experimentally using rapid prototyping. The explicit grounding of all abstract concepts, using the environment model of the mechatronic system under development, results in the ability to formally reason about required properties and still retain the operational semantics needed for a rapid prototyping approach to the evaluation of expected emergent properties.

We plan to evaluate the proposed concepts by means of a rapid simulation and prototyping extension for the Fujaba CASE tool and a series of alternative designs for large-scale scenarios of the RailCab case study. In its presented form, the approach exploits some domain specific characteristics of mechatronic systems. We also plan, however, to extend the approach to purely virtual systems as well, even though clearly none of their elements are physical. Those elements of the system that possess very immediate semantics and are directly accessible may be interpreted as a ”physical” domain model. Due to the proposed grounding, the approach offers a solid unambiguous base for formalizing the semantics of the more abstract concepts, and we believe that it can be employed with all the advantages it offers for mechatronic systems.

References

1. Dawson, D., D. Seward, D.B., Burge, S.: Mechatronics and the Design of Intelligent Machines and Systems. Nelson Thornes (2000)
2. Kennedy, J., Eberhardt, R.C.: Swarm Intelligence. Morgan Kaufmann publishers Inc.: San Mateo, CA, USA (2001)
3. Mullin, M.: Rapid prototyping for object oriented systems. Addison-Wesley, Reading (1990)
4. Brooks, R.A.: Intelligence Without Reason. In Myopoulos, J., Reiter, R., eds.: Proceedings of the 12th International Joint Conference on Artificial Intelligence (IJCAI-91), Sydney, Australia, Morgan Kaufmann publishers Inc.: San Mateo, CA, USA (1991) 569–595
5. Fischer, T., Niere, J., Torunski, L., Zündorf, A.: Story Diagrams: A new Graph Rewrite Language based on the Unified Modeling Language. In Engels, G., Rozenberg, G., eds.: tagt6. LNCS 1764, Springer (1998)
6. Rozenberg, G., ed.: Handbook of Graph Grammars and Computing by Graph Transformation : Foundations. World Scientific Pub Co (1997) Volume 1.
7. Singh, M.P.: On Competitive On-Line Algorithms for the Dynamic Priority-Ordering Problem. IEEE Computer **31** (1998) 40–47
8. Giese, H., Tichy, M., Burmester, S., Schäfer, W., Flake, S.: Towards the compositional verification of real-time uml designs. In: Proc. of the European Software Engineering Conference (ESEC), Helsinki, Finland, ACM Press (2003)
9. Giese, H., Vilbig, A.: Separation of Non-Orthogonal Concerns in Software Architecture and Design. Technical Report tr-ri-03-238, University of Paderborn, Paderborn, Germany (2003)
10. Giese, H., Burmester, S., Klein, F., Schilling, D., Tichy, M.: Multi-Agent System Design for Safety-Critical Self-Optimizing Mechatronic Systems with UML. In: OOPSLA 2003 - 2nd International Workshop on Agent-Oriented Methodologies, Anaheim, CA, USA. (2003)

11. Giese, H., Vilbig, A.: Separation of Non-Orthogonal Concerns in Software Architecture and Design. Software and System Modeling (SoSyM) (2005) (accepted).
12. Varró, D.: Automated formal verification of visual modeling languages by model checking. Journal of Software and Systems Modelling (2003) Accepted to the Special Issue on Graph Transformation and Visual Modelling Techniques.
13. R.Heckel, J.Küster, G.Taentzer: Towards automatic translation of UML models into semantic domains. In: Proceedings of the Applied Graph Transformation (AGT2002) Workshop. (2002) 11 – 22
14. Köhler, H., Nickel, U., Niere, J., Zündorf, A.: Integrating UML Diagrams for Production Control Systems. In: Proc. of the 22nd International Conference on Software Engineering (ICSE), Limerick, Irland, ACM Press (2000) 241–251
15. Dijkstra, E.W.: A Discipline of Programming. Prentice Hall, Englewood Cliffs, N.J. (1976)
16. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.M., Irwin, J.: Aspect-Oriented Programming. In: Proceedings of the European Conference on Object-Oriented Programming (ECOOP). Number 1241 in LNCS, Springer Verlag (1997)
17. Harrison, W., Ossher, H.: Subject-oriented programming (a critique of pure objects). In: OOPSLA'93. Volume 28 of ACM SIGPLAN Notices. (1993) 411–428
18. Clarke, S., Harrison, W., Ossher, H., Tarr, P.: Subject-Oriented Design: Towards Improved Alignment of Requirements, Design and Code. In: Conference on Object-Oriented Programming, Systems, Languages, and Applications, November 1-5, 1999, Denver, Colerado, USA. (1999) 325–339
19. Reenskaug, T., Wold, P., Lehene, O.A.: Working with Objects: The OOram Software Engineering Method. Addison-Wesley/Manning (1996)
20. Mekerke, F., Georg, G., Franc, R.: Tool Support for Aspect-Oriented Design. In: Proceedings of the Workshops on Advances in Object-Oriented Information Systems (OOIS 2002), Montpellier, France. Volume 2426 of Lecture Notes in Computer Science., Springer Verlag (2002) 280 – 289
21. Garcia, A., Silva, V., Chavez, C., Lucena, C.: Engineering multi-agent systems with aspects and patterns. J. Braz. Comp. Soc. **8** (2002) 57–72
22. Ferber, J., Gutknecht, O., Michel, F.: From Agents to Organizations: An Organizational View of Multi-agent Systems. In: Agent-Oriented Software Engineering IV, 4th International Workshop, AOSE 2003, Melbourne, Australia, July 15, 2003, Revised Papers. Volume 2935 of Lecture Notes in Computer Science., Springer Verlag (2003) 214–230
23. Deppe, M., Robrecht, M., Zanella, M., Hardt, W.: Rapid prototyping of real-time control laws for complex mechatronic systems. In: Proc. of the 12th IEEE International Workshop on Rapid System Prototyping (RSP 2001), 25-27 June 2001, Monterey, CA, USA, IEEE Computer Society (2001) 188–193
24. Schupp, G., Jaschinksi, A.: Virtual prototyping: the future way of designing railway vehicles. International Journal of Vehicle Design **22** (1999) 93–115
25. Beck, K.: Extreme Programming Explained: Embrace Change. Addison-Wesley Professional, Reading (1999)
26. Connell, J., Shafer, L.: Object-Oriented Rapid Prototyping. Yourdon Press, Englewood Cliffs, NJ (1995)
27. Popovici, A., Gross, T., Alonso, G.: Dynamic weaving for aspect-oriented programming. In: Proceedings of the 1st international conference on Aspect-oriented software development, ACM Press (2002) 141–147

Author Index

Agha, Gul 236
Ahmed, Amr 236
Alencar, Paulo 52

Bartolini, Claudio 213
Bastos, Lúcia R.D. 85
Bastos, Ricardo Melo 19
Blois Ribeiro, Marcelo 19

Castro, Jaelson F.B. 85
Choren, Ricardo 198
Cossentino, Massimo 36

de Barros Costa, Evandro 162
Dias da Silva, Leandro 162
Do, T. Tung 70

Faulkner, Stéphane 70

Garcia, Alessandro 52, 121
Giese, Holger 272

Hameurlain, Nabil 180
Henderson-Sellers, Brian 1
Holvoet, Tom 104

Jang, Myeong-Wuk 236
Jennings, Nicholas R. 213
Julien, Christine 254

Klein, Florian 272
Kolp, Manuel 70
Kulesza, Uirá 52, 121

Lucena, Carlos 52, 121, 198

Oliveira de Almeida, Hyggo 162

Payton, Jamie 254
Perkusich, Angelo 162
Preist, Chris 213

Roman, Gruia-Catalin 254

Seidita, Valeria 36
Shan, Lijun 144
Sibertin-Blanc, Christophe 180
Steegmans, Elke 104

Weyns, Danny 104

Zhu, Hong 144