Carlos Lucena  Alessandro Garcia
Alexander Romanovsky  Jaelson Castro
Paulo Alencar (Eds.)

Software Engineering for Multi-Agent Systems II

Research Issues
and Practical Applications

Springer
Volume Editors

Carlos Lucena
Alessandro Garcia
Pontifical Catholic University of Rio de Janeiro, Computer Science Department
Rua Marquês de São Vicente, 225, Rio de Janeiro, RJ 22453-900, Brazil
E-mail: {lucena, afgarcia}@inf.puc-rio.br

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

Jaelson Castro
Universidade Federal de Pernambuco, Centro de Informatica
Av. Prof. Luis Freire S/N, Recife, PE 50732-970, Brazil
E-mail: jbc@cin.ufpe.br

Paulo Alencar
University of Waterloo, School of Computer Science, Computer Systems Group
Waterloo, Ontario N2L 3G1, Canada
E-mail: palencar@csg.uwaterloo.ca

Cataloging-in-Publication Data applied for
A catalog record for this book is available from the Library of Congress.

Bibliographic information published by Die Deutsche Bibliothek
Die Deutsche Bibliothek lists this publication in the Deutsche Nationalbibliografie;
detailed bibliographic data is available in the Internet at <http://dnb.ddb.de>.

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

ISSN 0302-9743

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-Verlag. Violations are
liable for prosecution under the German Copyright Law.

Springer-Verlag is a part of Springer Science+Business Media
springeronline.com

© Springer-Verlag Berlin Heidelberg 2004
Printed in Germany
Typesetting: Camera-ready by author, data conversion by Olgun Computergrafik
Printed on acid-free paper        SPIN: 10991701        06/3142        5 4 3 2 1 0
Advances in networking technology have revitalized the investigation of agent technology as a promising paradigm for engineering complex distributed software systems. Agent technology has been applied to a wide range of application domains, including e-commerce, human-computer interfaces, telecommunications, and software assistants. Multi-agent systems (MASs) and their underlying theories provide a more natural support for ensuring important properties such as autonomy, mobility, environment heterogeneity, organization, openness, and intelligence. As a consequence, agent-based systems are likely to provide new approaches to dealing with the complexity of developing and maintaining modern software. However, developing robust large-scale agent-based systems will require new software engineering approaches. There are currently many methods and techniques for working with individual agents or with systems built using only a few agents. Unfortunately, agent-based software engineering is still in its infancy and existing software engineering approaches are unable to cope with large MASs.

The complexity associated with a large MAS is considerable. When a huge number of agents interact over heterogeneous environments, various phenomena occur which are not as easy to capture as when only a few agents are working together. As the multiple software agents are highly collaborative and operate in networked environments, they have to be context-aware and deal with environment uncertainty. This makes their coordination and management more difficult and increases the likelihood of exceptional situations, such as security holes, privacy violations, 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. However, many 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. Commercial success for MAS applications will require scalable solutions based on software engineering approaches in order to ensure effective deployment and to enable reuse.

The papers selected for this book represent advances in software engineering approaches to the development of realistic multi-agent systems. Research presented in this volume illustrates a broad range of techniques and methods that are being 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 demonstrated through examples that are representative of real-world applications. These papers describe experience and techniques associated with large MASs in a variety of problem domains.

Given the comprehensive selection of case studies and software engineering solutions for MAS applications, this book provides a valuable resource for a vast audience of readers. The intended primary audience for this book includes re-
searchers and practitioners who want to keep up with the progress of software engineering for 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 consist of: (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 five parts, which deal with topics related to: (i) requirements engineering, (ii) software architecture and design, (iii) modelling, (iv) dependability, and (v) MAS frameworks.

This book is a natural continuation of a previous one\textsuperscript{1}. The main motivation for producing this book was the 2nd International Workshop on Software Engineering for Large-Scale Multi-agent Systems (SELMAS 2003)\textsuperscript{2}, organized in association with the 25th International Conference on Software Engineering, held in Portland, Oregon, USA, in May 2003. SELMAS 2003 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 workshop participants to write chapters for this book based on their original position papers, and inviting other leading researchers in the area to prepare additional chapters. Following an extensive reviewing process involving more than 80 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 crossfertilization among individuals in this research area. It is only natu-


ral 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 that can be found in this book are only the beginning. Our thanks go to all our authors, whose work made this book possible. Many of them also helped during the reviewing process. We would like to express our gratitude to Juris Hartmanis, and Alfred Hofmann of Springer-Verlag for recognizing the importance of publishing this book. In addition, we would like to thank the members of the Evaluation Committee who were generous with their time and effort when reviewing the submitted papers. We gratefully acknowledge the support and cooperation of Cláudio Sant’Anna (LES, PUC-Rio) who helped us in the preparation of this volume.

December 2003
Carlos Lucena
Alessandro Garcia
Alexander Romanovskiy
Jaelson Castro
Paulo Alencar
Evaluation and Program Committee

P. Alencar (University of Waterloo, Canada)
A. Amandi (UNICEN University, Argentina)
B. Bauer (University of Augsburg, Germany)
F. Bergenti (University of Parma, Italy)
B. Blake (Georgetown University, USA)
L. Boloni (University of Central Florida, USA)
K. Breitman (PUC-Rio, Brazil)
P. Bresciani (ITC-irst, Italy)
J.-P. Briot (Laboratoire d’Informatique de Paris 6, France)
J. Castro (UFPE, Brazil)
R. Cost (University of Maryland, Baltimore County, USA)
E. Costa (UFAL, Brazil)
D. Cowan (University of Waterloo, Canada)
L. Cysneiros (York University, Canada)
J. Debenham (University of Technology, Australia)
P. Devarbu (University of California, USA)
A. Diaz-Pace (UNICEN University, Argentina)
M. d’Inverno (University of Westminster, UK)
C. Dony (LIRMM, France)
M. Endler (PUC-Rio, Brazil)
M. Fontoura (IBM Almaden Research Center, USA)
M. Fredriksson (Blekinge Inst. of Technology, Sweden)
A. Garcia (PUC-Rio, Brazil)
M.-P. Gervais (Laboratoire d’Informatique de Paris 6, France)
J. Giampapa (Carnegie Mellon University, USA)
P. Giorgini (University of Trento, Italy)
M. Gleizes (University Paul Sabatier, France)
H. Gomaa (George Mason University, USA)
M. Griss (University of California, Santa Cruz, USA)
Z. Guessoum (Laboratoire d’Informatique de Paris 6, France)
O. Gutknecht (LIRMM, France)
B. Henderson-Sellers (UTS, Australia)
T. Holvoet (Katholieke Universiteit Leuven, Belgium)
M. Huhns (University of South Carolina, USA)
E. Huzita (UEM, Brazil)
C. Iglesias (Technical University of Madrid, Spain)
C.M. Jonker (Vrije Universiteit Amsterdam, The Netherlands)
L. Kendall (Monash University, Australia)
R. Kessler (University of Utah, USA)
M. Klein (MIT, USA)
D. Kung (University of Texas, USA)
Y. Labrou (Fujitsu Laboratories of America, USA)
J. Leite (PUC-Rio, Brazil)
R. de Lemos (University of Kent, UK)
Evaluation and Program Committee

J. Lind (AgentLab, Germany)
C. Lucena (PUC-Rio, Brazil)
T. Maibaum (King’s College London, UK)
R. Mailler (University of Massachusetts, USA)
J. Maldonado (USP, Brazil)
M. Mamei (Univ. of Modena and Reggio Emilia, Italy)
D. Marinescu (Central Florida University, USA)
R. Milidiú (PUC-Rio, Brazil)
N. Minsky (Rutgers University, USA)
E. Oliveira (University of Porto, Portugal)
A. Omicini (University of Bologna/Cesena, Italy)
V. Parunak (Altarum Institute, USA)
A. Perini (ITC-irst, Italy)
O. Rana (Cardiff University, UK)
A. Rashid (Lancaster University, UK)
A. Romanovsky (University of Newcastle, UK)
G. Rossi (Universidad Nacional de La Plata, Argentina)
C. Rubira (UNICAMP, Brazil)
B. Schulze (LNCC, Brazil)
A. Staa (PUC-Rio, Brazil)
M. Stal (Siemens, Germany)
S. Stobart (University of Sunderland, UK)
K. Sycara (Carnegie Mellon University, USA)
A. Tripathi (University of Minnesota, USA)
W. Truszkowski (NASA, USA)
G. Wagner (Eindhoven Univ. of Technology, The Netherlands)
G. Weiss (Technical University of Munich, Germany)
M. Weiss (Carleton University, Canada)
E. Yu (University of Toronto, Canada)
F. Zambonelli (Univ. of Modena and Reggio Emilia, Italy)
A. Zisman (City University, UK)

Additional Reviewers

E. Belloni (UNICEN University, Argentina)
R. Choren (PUC-Rio, Brazil)
A. Helleboogh (Katholieke Universiteit Leuven, Belgium)
M. Kirby (University of Utah, USA)
T. Murata (Rutgers University, USA)
E. Olympiew (George Mason University, USA)
V. Silva (PUC-Rio, Brazil)
S. Till (Eindhoven Univ. of Technology, The Netherlands)
C. Urtado (École des Mines d’Alès, France)
D. Weyns (Katholieke Universiteit Leuven, Belgium)
Table of Contents

Requirements Engineering

The Agent at the Center of the Requirements Engineering Process ........ 1
Paolo Bresciani and Paolo Donzelli

Lexicon Based Ontology Construction ........................................ 19
Karin Koogan Breitman and Julio Cesar Sampaio do Prado Leite

Multi-agent Systems and Security Requirements Analysis ............... 35
Paolo Bresciani, Paolo Giorgini, Haralambos Mouratidis,
and Gordon Manson

Software Architecture and Design

Separation of Concerns in Multi-agent Systems: An Empirical Study .... 49
Alessandro Garcia, Cláudio Sant’Anna, Christina Chavez,
Viviane Torres da Silva, Carlos J.P. de Lucena, and Arndt von Staa

Architecting the Design of Multi-agent Organizations
with Proto-frameworks .......................................................... 73
Andrés Díaz Pace, Marcelo Campo, and Alvaro Soria

A Basic Taxonomy for Role Composition ..................................... 92
Elke Steegmans, Kurt Schelfhout, Tom Holvoet, Yolande Berbers,
Paul Valckenaers, and Bart Saint Germain

Modeling

Object-Oriented Modeling Approaches

to Agent-Based Workflow Services .......................................... 111
M. Brian Blake and Hassan Gomaa

Using the MAS-ML to Model a Multi-agent System ..................... 129
Viviane Torres da Silva, Ricardo Choren, and Carlos J.P. de Lucena

Software Engineering Challenges for Mutable Agent Systems ........ 149
Ladislau Bölöni, Majid Ali Khan, Xin Bai, Guoqiang Wang,
Yongchang Ji, and Dan C. Marinescu

Dependability

Improving Exception Handling in Multi-agent Systems .................. 167
Frédéric Souchon, Christophe Dony, Christelle Urtado,
and Sylvain Vauttier
On Manageability and Robustness of Open Multi-agent Systems ........ 189
   Naftaly H. Minsky and Takahiro Murata

Security Mechanisms for Mobile Agent Platforms
Based on SPKI/SDSI Chains of Trust ........................................ 207
   Michelle S. Wangham, Joni da Silva Fraga, Rafael R. Obelheiro,
   Galeno A. Jung, and Elizabeth Fernandes

MAS Frameworks

Farm: A Scalable Environment for Multi-agent Development
and Evaluation ........................................................................... 225
   Bryan Horling, Roger Mailler, and Victor Lesser

Role-Based Approaches for Engineering Interactions
in Large-Scale Multi-agent Systems ............................................. 243
   Giacomo Cabri, Luca Ferrari, and Franco Zambonelli

Evaluating Agent Architectures: Cougaar, Aglets and AAA .......... 264
   Ian Gorton, Jereme Haack, David McGee, Andrew Cowell,
   Olga Kuchar, and Judi Thomson

Author Index .................................................................................. 279
Abstract. Adopting Requirements Engineering (RE) techniques based on the fundamental notions of the agent-oriented programming paradigm, i.e., Agent, Goal, and Intentional Dependency, has been recognized as a crucial step towards a more homogeneous and natural software engineering process for complex socio-technical systems, among which Multi Agent Systems. The availability of simple representational tools is a key factor to guarantee stakeholders’ active involvement during RE, and therefore the success of the RE process itself. The paper introduces an agent-based Requirements Engineering Framework (REF), devised to deal with socio-technical systems, and support stakeholders’ participation. REF is designed around the adoption of a simple, but effective, graphical notation. Nevertheless, the simplicity of the graphical language may constraint the analysis process, reducing its flexibility and efficiency. This trade-off is carefully analysed, and some extensions are proposed, which do not affect REF clarity and intuitiveness, while enhancing REF capability to support requirements engineers in planning and implementing their analysis strategies.

1 Introduction

Agent-oriented programming paradigms and Multi Agent Systems (MAS) offer the great advantage of adopting the concept of Agent (together with the related notions of Goal, Intentional Dependency, Resource, and Task) commonly used also for describing the social setting in which the system has to operate. Thus, the possibility of using such notions as primitive elements for describing the system requirements during the Requirement Engineering (RE) process is becoming more and more relevant, especially for MAS application solutions. Agent- and goal-based RE approaches represent a first step to fill the gap between RE and Software Engineering (SE), especially, although not only [21], when MAS are concerned [5, 4]. Many SE techniques have been developed for MAS [18, 23], but few of them try to analyze and take into account the impact of the final system over the encompassing organization since the early phases of the RE process.

In such a perspective, the paper introduces an agent-based RE Framework (REF), initially applied for the definition of the requirements of a complex simulation environment [11]. REF is strongly based on diagrammatic notations, which immediately
convey the dependencies among different actors (or agents), and allow for a detailed analysis of the goals, upon which the actors depend, through a goal decomposition process. REF actors may be social (individuals and organizations, e.g., enterprises, departments, offices), as well as artificial (physical, as an hardware system, and software, as a software agent or a society of software agents inside a software system). The adopted graphical notation is widely inspired by the $i^*$ framework [25] for RE [24], and business analysis and re-engineering [26].

Denotationally inspired by $i^*$, REF introduces a clear process to drive requirements discovery, definition, refinement, and reconciliation [11], while maintaining the notational ingredients at an essential level. This choice, apparently constraining, results instead to be quite successful in practical terms. Several case studies [13, 12, 11] demonstrate, in fact, that the simplified notation facilitates the acceptance of the diagrammatic language by the stakeholders, and contributes to a quicker introduction of REF in the RE process. Nevertheless, such simplifications may still be consider as possible limits to the REF expressive power and, consequently, to the intentional analyses that could otherwise be carried out.

In this paper, after a brief introduction to REF (Section 2), an extensive and concrete case study is adopted to critically revise the analysis process underlying the current framework, and propose two notational and methodological extensions as solutions (Section 3). In Section 4, we suggest that, when appropriately used, the proposed extensions do not jeopardize REF usability and its acceptance by the stakeholders, but support the analysts in planning and performing a more efficient analysis strategy.

2 Introduction to REF

REF is designed to deal with, and reason about, socio-technical systems. Here the software system and its application context form a larger human and technological system that has to be treated and analyzed as a whole, and the overall needs are the ones to be fulfilled [15, 18, 23]. Complexity of socio-technical systems goes beyond working procedures and the complexity of the software system itself: it encompasses the complexity generated by the impact of the system upon the organizational structure, from the business process, to the behavior of the single employee. The basic idea behind REF is to provide the analyst with the right tools to capture the high-level organizational needs and transform them into organizational and system requirements. Application context has in fact often to be adapted in order to exploit the capabilities of the new system. The framework tackles the modeling effort by breaking the activity down into more intellectually manageable components, and by adopting a combination of different approaches, on the basis of a common conceptual notation.

Agents are used to model the organization [9, 11, 19, 24]. The organizational context is modeled as a network of interacting agents, collaborating or conflicting in order to achieve both individual and organizational goals. Goals [22, 1, 8, 11, 24] are used to model agents’ relationships, and, eventually, to link organizational needs to system requirements. According to the nature of a goal, a distinction is made between hard goals and soft goals. A goal is classified as hard when its achievement criterion is sharply defined. For example the goal “document be available” is a hard goal, being easy to
check whether or not it has been achieved (i.e., is the document available, or not?). For a soft goal, instead, it is up to the goal originator, or to an agreement between the involved agents, to decide when to consider the goal as achieved. For example, the goal “document easily and promptly available” is a soft goal, given that as soon as we introduce concepts such as “easy” and “prompt”, different persons usually have different opinions.

Distinguishing goal modeling from organizational modeling, and then further distinguishing between hard goal modeling and soft goal modeling, is a key aspect of REF, and helps reducing the complexity of the modeling effort. The proposed framework, therefore, supports three inter-related modeling efforts [3]: the Organizational Modeling, the Hard Goal Modeling and the Soft Goal Modeling.

During Organization Modeling, the organizational context is analyzed and the agents and their goals identified. Any agent may generate its own goals, may operate to achieve goals on the behalf of some other agents, may decide to collaborate with or delegate to other agents for a specific goal, and might clash on some other ones. The resulting goals will then be refined, through interaction with the involved agents, by hard and soft goal modeling.

The Hard Goal Modeling seeks to determine how an agent thinks to achieve a hard goal, by decomposing it into more elementary subordinate hard goals and tasks (where a task is a well-specified prescriptive activity).

The Soft Goal Modeling seeks to determine how an agent thinks to achieve a soft goal, by decomposing it into more elementary subordinate soft goals, hard goals, tasks, resources [24, 7], and constraints [11, 14]. Soft goal modeling aims at producing the operational definitions of the soft goals, thus soft goals refinement has to be reiterated until only hard goals, tasks, resources and constraints are obtained (that is, until all the “soft” aspects are dealt with).

Both soft and hard goals are refined by repetitively asking the agents what they needed to know, to perform, have delivered or have performed (and by whom, leading in this way to the identification of new agents), in order to consider the goal as achieved. For soft goal modeling, in addition, REF provides the constraints as mechanisms to harden softness. So, for example, the soft goal “document easily and promptly available”, besides spawning the hard goal “document available”, will lead also to a set of constraints (e.g., types of access channels, number of hours after which a document is available, etc.), specifying the concepts of easy and prompt.

2.1 The Case Study: ERMS

In order to introduce REF, paving at the same time the way for discussing its limits and possible extensions (Section 3), we describe an on-going project aiming at introducing an Electronic Record Management System (ERMS) into a complex administrative organization. The system is expected to affect a large community of knowledge workers. Indeed, ERMS is at the moment used by more than 300 employees and handles a flow of about 200,000 documents/year, but it is designed to reach about 2000 users and 2 million documents/year.

An ERMS is a complex Information and Communication Technology (ICT) system that allows efficient storage and retrieval of document-based unstructured information,
by combining classical filing strategies (e.g., classification of documents on a multi-level directory, cross-reference between documents, etc.) with modern information retrieval techniques. It usually provides mechanisms for facilitating routing and notification of information/document among the users, and interacting with similar (typically remote) systems, through e-mail and XML. An ERMS represents the basic element for a knowledge workplace, i.e., a working environment where a knowledge worker can easily access and gather information, produce knowledge and deliver results through a multitude of channels (from personal computers, to laptops, PDAs, mobile phones, etc.). It is, in fact, a necessary step for introducing more sophisticated document management tools, such as workflow technology and digital signature, both fundamental mechanisms for a paperless and ubiquitous working environment. Several factors (international benchmarking studies, citizens demand, shrink budgets, etc.) lead to the decision of leveraging new technologies to transform the organization bureaucratic structure into a more creative, and knowledgeable environment.

Starting the Analysis. The initial organization model expressing the situation introduced above is shown in Figure 1. Circles represent Agents, and dotted lines are used to bound the internal structure of complex agents; that is, agents containing other agents. Thus, in Figure 1, the agent Organization Unit is a complex agent, representing the organizational structure into which the ERMS has to be introduced; the Head of Unit is the agent acting within the Organizational Unit who is in charge for achieving the improvement objectives (modeled by the soft goals exploit ICT to increase performance while avoiding risks, and cost/effective and quick solution). Goals, tasks and agents (see also next figures) are connected by dependency-links, represented by arrowhead lines. An agent is linked to a goal when it needs or wants that goal to be achieved; a goal is linked to an agent when it depends on that agent to be achieved. Similarly, an agent is linked to a task when it wants the task to be performed; a task is linked to an agent when the agent is committed at performing the task. Again, an agent is linked to a resource when
Fig. 2. The exploit ICT to increase performance while avoiding risks Soft Goal Model

it needs that resource; a resource is linked to an agent when the agent has to provide it. By combining dependency-links, we can establish dependencies among two or more agents.

At this point, to continue with the analysis, the goals resulting from the initial model have to be refined by means of goal modeling. In the following we focus on the soft goal exploit ICT to increase performance while avoiding risks.

**Goal Modeling.** Figure 2 describes how the soft goal exploit ICT to increase performance while avoiding risks is iteratively top-down decomposed to produce a set of tasks, hard goals, and constraints that precisely define it. Figure 2, in other terms, represents the strategy that the Head of Unit (as the result of a personal choice or of a negotiation with the upper organizational level) will apply to achieve the assigned goal. Again, the arrowhead lines indicate dependency links: a goal depends on a sub-ordinate goal, task or constraint, when it requires that goal, task or constraint to be achieved, performed, or implemented in order to be achieved itself. Goals decompositions may be conjunctive (all the sub-components must be satisfied, to satisfy the original soft goal), indicated by the label “A” on the dependency link, or disjunctive (it is sufficient that only one of the components is satisfied), indicated by the label “O” on the dependency link (see Figure 4). According to Figure 2, the Head of Unit has to increase personal performance, to increase productivity of the whole unit, and also to avoid risks due to new technology. Let us consider in details only the first sub - soft goal, i.e., increase personal performance. It spawns two subordinate soft goals, easy document access, for which the Head of Unit will require a multi-channel access system in order to be able to check and
transfer the documents to the employees also when away from the office (asking at the same time that secretary does not filter the documents), and increase process visibility, to take better informed decisions. In particular, the soft goal increase process visibility will eventually lead to the identification of some tasks (functionalities) that the system will have to implement to provide some data about the process (number of documents waiting) and about the employees (provide employee's number of documents), together with some associated constraints, represented by a rounded-rectangle with a horizontal line. In Figure 2, for example, they specify the frequency of update: daily update for the process data and weekly update for the employee’s ones. Finally, in Figure 2, the items in bold outline are those that the Head of Unit will pass out, having decided to depend on other agents for their achievement. For such a reason, they are not further analyzed; instead they will be refined as further agreement between the Head of Unit and the agent that will be appointed of their achievements.

**Extending the Organizational Model.** The results of the goal analysis allow us to enrich the initial organization model in Figure 1, leading to the model in Figure 3. Here – where some details have been omitted for the sake of clarity – some new agents have been introduced: the Archivist and the Employee, which have to be more productive, the IT (i.e., the Information Technology Unit), which has to guarantee security, and the
The Agent at the Center of the Requirements Engineering Process

**Fig. 4. The be more productive Soft Goal Model**

ERMS. From Figure 3, we can also see that the Head of Unit has decided to delegate the soft goal cost/effective and quick solution to the IT agent, which, on its turn, will have to achieve other goals coming from the external environment, such as, for example, apply public administration standards. Again, at this point, the analysis can proceed through goal modeling. Below, we focus on how the Employee will try to achieve the soft goal be more productive.

**More Goal Analysis.** In Figure 4 we can see how, in order to be more productive, the Employee will require a system easy to learn, and facilitating the collaboration with other employees easier (soft goal make collaboration easier).

The soft goal easy to learn will spawn, among other items here omitted, the constraint adopt known technologies (i.e., technologies which the employee is already used to), whereas the soft goal make collaboration easier will lead, through further refinement, to a serie of hard goals implying specific capabilities (e.g., either a teleconference or an IP-based collaboration tool) and access channels (e.g., mobile phone, laptop, etc.).

As already seen for the IT agent, also the Employee may have his/her own needs, leading to new goals. For example, the Employee may be concerned of possible side effects on the privacy related to the introduction of the ERMS, represented by the soft goal protect my privacy upon the ERMS, introduced in Figure 3. By modeling the soft goal protect my privacy (Figure 5), we can discover that the Employee wants to be sure that its own private area of work will not be accessible by others, asking for example to be enabled to adopt a cryptography tool (with a 128 bit length security key) to protect particular data, but above all, given that all the activities will be performed through the ERMS, that s/he does not want someone to be able to monitor his/her actions. This clearly conflicts with the possibility of monitoring the employee’s performance that was required by the Head of Unit (Figure 2), requiring some trade-off to be identified.
2.2 REF Background

In the last years, domain modeling has been recognized in RE as crucial for a successful system development: this includes activities aiming at understanding why the system is needed, how it would meet such goals, and possible alternatives, while taking into account the interests and the perspectives of all the different stakeholders [1, 4, 10, 22, 24]. REF builds on such results by combining advanced requirements engineering techniques (i.e., agents [9, 11, 24] and goals [1, 8, 11, 19, 22, 24]) with software quality modeling approaches [2, 6, 17], to capture the stakeholders’ perception of quality since the beginning of a new project, and to produce agreeable-upon and implementable functionalities and constraints.

REF is strongly based upon $i^*$, the modeling framework suggested by Eric Yu [26, 25, 24]. However, it introduces some simplifications and adopts a more pragmatic approach in order to obtain a greater and more active involvement of the stakeholders during the requirements discovery, elicitation and formalization process. The aim is to let the stakeholders to easily grasp the notation since the beginning of a project. Thus, the adopted simplifications include: 1) the use of only one type of actor/agent (i.e., no distinction is made between agent, role and position); 2) the use of only one kind of link, both as dependency link as well as decomposition and means-end link. This choices allows for a more intuitive reading of the REF diagrams, introduces the possibility of easily modeling many-to-one and/or one-to-many agent-goal relationships very common in real situations, as well as a more natural representation of the flow of dependencies, from the dependencies among actors, in the organization model, to the decomposition dependencies, in the (hard and soft) goal models.

For what concerns the application process, REF adopts a strict top-down approach, during which the analysts, in close cooperation with the stakeholders, drill through the organization, until reaching the desired level of detail, by using three different (but similar) modeling tools: organization models, soft goal models, and hard goal models. In
particular, once an initial model of the organization is built, the REF process evolves in a cyclic way through two main phases: a) goal modeling phase, during which the soft and hard goals discovered during organization modeling are refined; and b) organization modeling phase, during which the analysts use the information gained modeling the goals to enrich and extend the initial organizational model: i.e., to replace the goals with their models, and to introduce the new agents identified as relevant to achieve those goals. New agents are therefore identified, and brought into the picture, depending on the specific goals to achieve, and the indications of the other agents. New agents usually lead to new goals, triggering the goal-modeling phase again [3]. Such a cycle is continued until the desired (and needed) level of details is reached. Unlike i* (and also NFR [7] and Tropos [20, 4], that share with i* the notion of soft goal satisficement [24]), REF emphasizes the operational role that soft goals can play in deriving the requirements of a new system. Soft goals, in fact, beyond being crucial in supporting reasoning between alternatives (as in i*, NFR and Tropos), since the early stages of a project, can also provide a systematic and organized way of handling non-functional requirements. In particular, REF recognizes the need of explicitly “resolving” soft goals, by turning them into more manageable (and implementable) hard goals, tasks and constraints. This last notion – constraint – is not, indeed, present in i*, NFR, nor Tropos.

To refine a soft goal, the analysts have first to specify the actions that are usually implied, and sometimes hidden in the goal (e.g., to make a document available in the soft goal document easily and promptly available), then they have to make operational its softness (e.g., in identifying the constraints that describe the concepts of easily and promptly). By repetitively asking the agents what they need to achieve, to perform, or have delivered in order to consider the soft goal as achieved, it is possible to slowly separate the actions implicit in the goal (represented by the emerging hard goals, tasks, and resources), from its softness (represented by the emerging and more elementary soft goals). Eventually, the emerging soft goals will be “pure” soft goals, specifying quality attributes of the identified actions. To further refine these pure soft goals, REF borrows ideas from quality modelling and measurement definition techniques. In refining such a softness, i.e., to make it operational, in fact, the analysts perform a two-step process: 1) First, they build a quality model [6] of the soft (quality) attribute, by identifying the characteristics that the stakeholders assume to be important in judging it. For example, for assessing how easily and promptly a document is available, we can identify, as relevant, system characteristics such as the types of access channels, the number of hours after which the document is available, the possibility of mobile access, and so on. 2) Second, they populate the quality model and freeze the result, by specifying for each characteristic the desired values (or range of values), according to the corresponding measurement scale [14]. So, for example, for the limit of hours after which the document has to be available (absolute scale), a number or a range of numbers have to be assigned, while, for the types of access channels (nominal scale), the desired values have to be specified (e.g., mobile phone, laptop, etc).

To define the quality model of the soft attribute, the analysts may turn to quality models already available in literature [14], such as the McCall, the Bohem, or the IEEE standard, or adopt more sophisticated empirical measurement identification methods, such as the Goal Question Metric (GQM) approach proposed by Basili [2] and its ex-
tensions [6]. Quality models show little flexibility and may result difficult to customize to the specific contexts and agents’ needs. On the contrary, empirical measurement identification methods recognize that quality issues cannot live in isolation, but have to be derived from, and linked to, their operational context (i.e., stakeholders, problem domain, underlying reasons, etc.).

The most classical of the empirical methods is the GQM approach, based on the idea that measures have to be identified starting out from the analysis of the goal that has to be achieved. The goal (precisely stated, by specifying the object of study, the purpose, the quality focus, the viewpoint and the context) is refined into several questions that break down the issue into its major components, and each question is refined into metrics. Basically, measures are obtained by applying a question-answer mechanism: ask which “Questions” we should be able to answer in order to achieve the “Goal”, and then ask what “Metrics” we should collect to be able to answer those questions. In such a perspective, a GQM-like approach has been applied for goal refinement during soft goal modeling. For each soft goal, in fact, REF clearly identifies the target object (object of study), the reasons (purpose), the soft attribute (quality focus), the stakeholder (viewpoint), and the application context (context), providing the basis upon which a GQM-like question-answering mechanism can be adopted as technique to support elicitation. In other terms, to support the identification and to populate the quality model, i.e., the set of constraints (metrics) that complete the goal refinement. Constraints do not leave in isolation, but assume meaning only when associated to the hard goal, task, or resource that they specify.

3 Improving REF Analysis Capability

As described in Section 2, REF aims at providing a representational framework for requirements discovery, definition and analysis characterized by a graphical notation sufficiently expressive to deal with complex problems but, at the same time, simple enough to be easily and quickly grasped by the stakeholders, usually unfamiliar with RE techniques. These are quite relevant aspects which make REF suitable for real applications [13, 12, 11], and ensure an active stakeholders’ involvement, critical for a quicker and more effective knowledge acquisition and requirements elicitation process capable of leading to domain descriptions close to the real state of affairs [22, 11].

We believe REF simplicity is mainly due to two key features, characterizing its goal modeling activities: 1) a very sparse notation, and in particular the use of only one type of link (the dependency link); 2) the focus, during goal analysis, on only one goal and one agent at time. This leads to simple tree-like goal diagrams, generated by a strict top-down process, and independent from the way in which they are developed. In other terms, while both these REF aspects lead to easily readable diagrams, the second one, in particular, allows the analysts and the stakeholders to focus on only one problem at the time, without need to worry about the order in which the analysis of different sub-trees is carried out: the resulting diagram is always generate in the same shape, whichever node expansion sequence is followed (breath first, depth-first, o mixed).

Although important for REF simplicity, such features may however constraint the analysis process, limiting its flexibility and efficiency. Tree-like diagrams, in fact, de-
Developed by focusing on one agent and one goal at the time, may oversimplify real case situations and reduce the possibility for the analysts of efficiently dealing with situations in which parts of the diagram (sub-goals or constraints, or tasks, or resources) could be in common or in conflict within the diagram itself or between different ones. In other terms, they hinder the analysts’ capabilities of identifying, reasoning about, and respectively, resolving or exploiting sharing and conflicts situations. As discussed in [11], dealing with tree only requires to perform most of the analysis before being able to detect conflict or sharing situations, and then re-do part of the effort.

In the following, we highlight and discuss concrete examples where such limits appear to be evident, and suggest extensions of REF, in order to introduce more flexibility in the process of model description and requirements elicitation, without affecting its initial simplicity.

3.1 Sharing Goals (Tasks, Constraints, …)

Top-down tree expansion and analysis induces to introduce different sub-goals (or constraints, or tasks, or resources) for any different goal that is found during the goal analysis activity, even when different goals could be (partly) satisfied by the same sub-goal (or constraint, or task, or resource). This situation may be further distinguished in at least three different sub-cases.

First Case: Commonalities within the Same Goal Diagram. In Figure 2, for example, two distinct constraints have been introduced for satisfying the soft goals provide process performance and provide employee’s performance, namely the constraints daily update and weekly update, also if the Head of Unit could have accepted them to share the same constraint (e.g., a twice a week update, as in Figure 6). According to REF, in fact, any sequence may have been followed in analyzing the two soft goals, and the two constraints may have been introduced in two very different moments, making it very difficult to spot that a common (although slightly different) constraint could have been adopted. This tradeoff, on the other side, could have been identified and judged as acceptable if considered by the analyst together with the Head of Unit at the proper moment during the elicitation activity.

While the difference between Figures 2 and 6 is minimal, regarding only leaf nodes (as highlighted by the dotted circle), let us consider a more complex hypothetical case, in which the two collapsing nodes are non-leaf nodes, but have deep trees expanding from them.

In this case, relevant parts of the two sub trees, rooted in the two nodes, would have to be revised, in order to consider an alternative non-tree-based analysis. It results therefore to be strategic to be able to introduce a common child for the two nodes before proceeding with the analysis of the nodes sub-trees. It is clear in fact how different diagram evolution strategies, and thus development sequences, could lead to quite different results or, eventually, to the same result but with a different degree of efficiency (depending on the strategy). For example, a top-down breath-first diagram expansion could be preferred to a top-down depth-first one. In this way, it may appear appropriate to develop a shared sub-tree only once, with two advantages: 1) at the elicitation level,
the analysis does not have to be carried out twice; 2) at the implementation level, the complexity of the system will be reduced, being two potentially different requirements, and all the following details, collapsed in one. Of course, not always merging two nodes (and eventually all their sub-trees) is a reasonable choice, and has to be the result of a compromise that needs to be carefully evaluated. It is up to the analyst (together with the stakeholders) to decide if such an option is acceptable. The important fact here is that, allowing the analysts to show the possible set of shared requirements (easily and visually appraisable) allows also to reason about the amount of savings in elicitation and implementation time. In other terms, the analysts have a more powerful tool to evaluate alternatives, also taking into account the possible consequences in terms of system design and development efforts. Considerable advantages seems to be introduced by the possibility of considering also directed a-cyclic graphs, instead of only trees, as design artefacts.

Indeed, a risk is present, and has to be carefully considered: the strict REF approach is much more simple, and make the tree expansion order irrelevant, so improving usability and comprehension of the process; to find, instead, possible common sub-goals,
continuous (or at least very frequent) comparison of sub-goals is necessary, and the expansion sequence may make some difference: this constraint make the process users (that are, the analysts, but specially the stakeholders) more bound to rules and intuition on the best path.

**Second Case: Commonalities within Different Goals Diagrams of the Same Actors.**
This is a specialization of the first case, where similar sub-goal sharing happens among goals placed upon an actor during the organizational model analysis. In this case, the REF methodology would lead the analyst and the stakeholder to duplicate the sub-goal in two different diagrams, usually with slightly different labels, although with the same semantics. Again, catching these cases would avoid duplicated efforts.

Moreover, as soon as a sub-goal is recognized as shared by different goal diagrams, immediately it becomes more relevant than other ones, being its satisfaction useful (or necessary or sufficient, or both) to two goals. Thus, the decision could be made that more elicitation, analysis and implementation efforts should be applied to this particularly valuable sub-goal.

**Third Case: Commonalities within Different Goals Diagrams of Different Actors.**
This is a more intriguing situation, where the same sub-goal is shared between two different actors, as a consequence of the independent analysis carried out by the two agents on two different goals placed upon them.

Consider for example the increase personal performance soft goal in Figure 2, from which the soft goals easy document access and increase process visibility are derived. The first one, on its turn, leads to the soft goal multi-channel access. The same soft goal multi-channel access is present also in Figure 4, as result of the analysis of the soft goal be more productive for the Employee, that is as a soft goal that the employee considers as relevant in order to become more productive. Again, as in the previous cases, the analysis of such a shared soft goal immediately assume a higher relevance and priority over the analysis of other ones. Two actors desire its satisfaction! The analysis can therefore be carried out once for both the actors, exploiting the approach to better combine their needs in a synergic way, and avoiding duplications that, although most of the times are due only to slightly different ideas, may generate over complexity in terms of systems functionalities. For example, leading to the selection of only one kind of mobile access channel able to satisfy both the agents.

From the analysis of all the previous three cases, clearly emerge the need of introducing in REF some mechanism to support the analysts during goal modeling. In particular, we propose to provide the analysts with a specific notation to be used to highlight situations where they believe that some commonalities could be hidden, i.e., that shared-goals could arise during the analysis. In other terms, to introduce in REF a notation suitable to act as a high-level reasoning support tool to enable the analysts to record their intuitions while building the goals models: i.e., making notes to drive their strategies. For example, to highlight where a top-down bread-first diagram expansion may be preferable to a top-down depth-first strategy. As such a notation, we introduce a dotted labeled line to connect the two or more goals among which a possible sharing could emerge. The line (the \textit{S-connection}) does not have arrows, being the relationship
perfectly symmetric, and it is marked by the label $S$, that stands for Sharing. Figure 7 shows a fragment of Figure 2 where the S-connection has been adopted. In particular, it shows how the S connection could have been used during the analysis of the soft goal to highlight in advance the possibility of sharing between the soft goals provide employees performance and provide process performance (the first example case previously analyzed). In the same way, in Figure 8 is depicted the use of the S-notation to highlight, within the soft goal analyzed in Figure 2, a possible sharing between the soft goal increase personal performance, that the Head of Unit wants to achieve, and the soft goal be more productive, that the Head of Unit imposes, transfers to, the Employee (the third example case previously analyzed).

It is worth noting how the S-notation is only a reasoning support mechanism that tends to disappear once the analysis proceeds. In other terms, the S-notations purpose is to mark possible sharing situations to drive the analysis (e.g., bread-first, multi-agents analysis, repeated back to back comparisons, and so on), but does not have any reason
to exist any more once the goals have been explored: the initial REF notation, with its simplicity, is sufficient for that regard.

3.2 Clashing Goals (Tasks, Constraints)

Up to here we have been concerned with the possibility of sharing (among different actors, or among goals of one actor) one or more sub-goals (or constraints, tasks, and resources). Another interesting, and actually very common situation, regards the possibility of efficiently dealing with clashing needs (goals, constraints, tasks, or resources). Of course, as far as a strictly top-down analysis is performed (as currently done in REF), there is no explicit reason to introduce contrasting elements in the design.

But they may arise as soon as considering the situations discussed above. As well as during a top-down analysis an introduced sub-goal may be recognized as helpful for another one (possibly of another actor), in a similar process it may result (possibly) harmful. In addition, during the analysis, new agents may have to be introduced into the context (e.g., the Head of Unit requires the Employee to be more productive), and such new agents may express their own needs by introducing new goals in the context. Such goals may very easily clash with other goals already in the context.

A clear example is represented by the soft goal protect my work performance in Figure 5, leading to the constraint that the ERMS should not be allowed to monitor and record data regarding the Employee actions. This clashes with the desired expressed by the Head of Unit of knowing the number of documents each Employee is dealing with (Figure 2). Indeed, as just seen in the discussed example, REF already supports the recognition of such situations. When fully operationalised in terms of tasks and constraints, goals models in fact can be adopted to detect clashing situations and to resolve them. In this case, for example, the decision could be made of providing the Head of Unit only the average of the documents dealt with by all the Employees, so protecting the privacy of the single one. Nevertheless, we think it may be very useful to be able to recognize such situations as early as possible, also at a very qualitative level, before than pushing the analysis down to the final constraints and details. For such a reason, to enable the analysts to mark possible conflicting situations (and build their refinement strategy to deal with them), we introduce the H-connection, H for Hurting. This is a powerful tool to detect possible conflicts and try to reconcile different stakeholders’ points of view, evolving the analyses only along the most promising alternatives.

An example application is shown in Figure 9, where the H-notation has been used to highlight the possibility of a conflict between two goals before proceeding in their analysis (i.e., the soft goal provide employees performance is not broken down into tasks before taking into account the protect my privacy one).

4 Conclusions

The paper introduced an agent-based Requirements Engineering Framework (REF), explicitly designed to support the analysts in reasoning about socio-technical systems, and transform high-level organizational needs into system requirements, while redesigning the organizational structure itself. The underlying concepts and the adopted notations
make of REF a very effective and usable tool, able to tackle complex real case situations, while remaining simple enough to allow an active stakeholders’ involvement.

However, we felt that REF could be improved to better support the analysts in dealing with complex system/organizational design issues, such as shared and clashing stakeholders’ needs. In both cases, an early detection of the problem can, in fact, lead to better analysis results: shared needs could be objects of a more intensive analysis effort to exploit commonalities to reduce complexity and increase reusability; clashing needs could be solved at a very early stage, to focus then the analysis only towards the most promising alternatives. Two graphical notations (i.e., S-connection and H-connection) have thus been introduced to allow the analysts to mark such situations and better reason about how to build their strategy. They are pure analyst-oriented tools that do not affect REF usability in terms of stakeholders’ perspective. We expect that their use can greatly improve the analysts’ capabilities of building the strategy to deal with shared and clashing interests, by allowing them to easily connect items (goals, constraints, tasks and resources) across different models and diagrams, to support their reasoning about the problems and about how to face them (e.g., whether or not perform a multi-agent elicitation session). At the same time, the S-connections and H-connections do not have to appear while interacting with the stakeholders, so that, once the analysis is completed, only the REF original notation is sufficient. Thus, S-connections and H-
connections have mainly to be considered as a tool for the elicitation process, and not as a support to alternatives evaluation, unlike what offered by other formalisms, as, e.g., NFR [7].

Finally, although REF addresses only the early stages of the RE process, the possibility of combining its outcome with techniques more suitable for dealing with further system development phases has been investigated. For example, practical results suggest that it can be usefully applied as a forerunner to both object-oriented approaches, such as those based upon UML [10], as well as agent-oriented approaches for MAS systems [5, 16, 20].

References


Lexicon Based Ontology Construction

Karin Koogan Breitman¹ and Julio Cesar Sampaio do Prado Leite¹, ²

¹ PUC-Rio, Computer Science Department, LES
Rua Marquês de São Vicente 225 - 22453-900, Rio de Janeiro, RJ, Brazil
karin@inf.puc-rio.br
http://www.inf.puc-rio.br/~julio
² On leave at Dept. of Computer Science, University of Toronto
10 King's College Road, Toronto, M5S3G4, Canada

Abstract. In order to secure interoperability and allow autonomous agent interaction, software for the web will be required to provide machine processable ontologies. Traditional deliverables of the software development process are the code, technical documentation, to support development and maintenance and use documentation, to provide user support. In the case of web applications, ontologies will also be part of the deliverables. Ontologies will allow machines to process and integrate Web resources intelligently, enable quick and accurate web search, and facilitate communication between a multitude of heterogeneous web-accessible agents [1]. We understand that the responsibility, not only for making explicit this requirement, but also to implement the ontology, belongs to software engineers. Currently the development of ontologies is more of a craft then a systematic discipline. We are proposing a process for the systematic construction of ontologies, centered on the concept of application languages. This concept is rooted on a representation scheme called the language extended lexicon (LEL). We demonstrate our approach using an example in which we implement a machine processable ontology for a meeting scheduler using the ontology language DAML+OIL.

1 Introduction

Researchers from industry and academia are exploring the possibility of creating a "Semantic Web," in which meaning is made explicit, allowing machines to process and integrate Web resources intelligently. This technology will allow interoperability among development of intelligent internet agents in large scale, facilitating communication between a multitude of heterogeneous web-accessible devices. Unfortunately, the majority of the information available is in a format understandable to humans alone, making the automation of search and retrieval processes very hard [1]. Ontologies should provide the necessary meaning to web content therefore enabling software agents to understand and retrieve information in relevant contexts [14].

The development of ontologies today is more of a craft than a science [10]. Available processes for ontology building provide little support to basic knowledge acquisition activities, such as elicitation and modeling, and focus on formalization aspects [15,31,9,13,29]. The final result is that the ontology construction process becomes time consuming and onerous.
From an agent perspective, an ontology is a software deliverable. It has to be made available in order to allow interoperability among a large community of agents that, in turn, interact with other ontologies. Agent developers need ontologies to test their agent behavior and to promote large scale interoperability. We propose an ontology construction process rooted on a representation scheme called the language extended lexicon (LEL) [23]. A lexicon term is composed of notion (denotation) and behavioral responses (connotation). The list of behavioral responses makes the LEL special, in relation to other lexicons or dictionaries, because it provides additional information to the meaning of terms in the format of a list of relationships to other lexicon terms. The particular structure of the LEL makes it an excellent basis for ontology construction.

The proposed process is systematic enough as to allow people who are not experts in knowledge engineering to perform it. We demonstrate our proposal using an example in which we implement a machine processable ontology for a meeting scheduler in the DAML+OIL ontology language.

2 Ontologies

We adopt the ontology structure $O$ proposed by Maedche [26]. According to the author, an ontology can be described by a 5-tuple consisting of the core elements of an ontology, i.e., concepts, relations, hierarchy, a function that relates concepts non-taxonomically and a set of axioms. The elements are defined as follows:

$$O := \{ C, R, \mathcal{H}, \text{rel}, \mathcal{A} \}$$ consisting of:

- Two disjoint sets, $C$ (concepts) and $R$ (relations)
- A concept hierarchy, $\mathcal{H}$: $\mathcal{H}$ is a directed relation $\mathcal{H} \subseteq C \times C$ which is called concept hierarchy or taxonomy. $\mathcal{H}(C_1, C_2)$ means $C_1$ is a subconcept of $C_2$
- A function $\text{rel}: R \rightarrow C \times C$ that relates the concepts non-taxonomically
- A set of ontology axioms $\mathcal{A}$, expressed in appropriate logical language.

Most existing ontology representation languages can be mapped to this structure. In the next section we survey some of these languages.

2.1 Ontology Implementation Languages

At first we considered the use of the Resource Framework Description (RDF) [19]. RDF allows to express the semantics of a web page in terms of metadata. It provides a set of primitives for modeling simple ontologies in RDF schema, e.g., “SubClassOf” and “SubPropertyOf”. RDF however, was criticized as an ontology language because it lacked expressiveness [17]. In the RDF Schema logical connectives such as negation, disjunction and conjunction are not provided, thus restricting the expressive power of the ontology.

We also considered using the Simple HTML Ontology Extension (SHOE) developed at the University of Maryland, prior to XML and RDF [25]. SHOE is an ontology-based knowledge representation language that is embedded in web pages. The underlying philosophy of SHOE is that intelligent internet agents will be able to better
perform their tasks if the most useful information is provided in a structured way. SHOE extends HTML with a set of knowledge oriented tags and associates meaning to the contents of a page by making each web page commit to one or more ontologies. The ontologies permit the discovery of implicit knowledge through the use of taxonomies and inference rules. Compared to RDF, SHOE is analogous an RDF Schema but with less expressive power [3], e.g., SHOE does not allow negation in the claim statement nor the subclass relationship for properties. Maintenance of ontologies is also an issue in SHOE, for they are embedded in the web pages as opposed to as separate objects.

The ontology inference layer (OIL) was sponsored by the European Community via the on-to-knowledge project. OIL sprung from the need of an expressive language for creating ontologies on the web, since RDF provides inadequate expressiveness and lacks formal semantics and reasoning support. OIL’s formal semantics and efficient reasoning support is provided by Description Logics. The semantics of OIL rely on a translation into the description logic $SHIQ$ extended with concrete data types, $SHIQ(d)$. A complete mapping of OIL to $SHIQ(d)$ is available in [20]. The OIL community made available tools that support editing and ontology reasoning. There are three editors available, OntoEdit, developed at the University of Karlsruhe, OILed, developed at the University of Manchester and Protégé-2000, developed at Stanford University [8]. Reasoning support for OIL is available and provided by the FaCT (fast classification of terminologies) inference engine, publicly available at http://www.cs.man.ac.uk/~horrocks/FaCT/ [20]. The reasoning services provided include inconsistency detection and determining subsumption relationships. OIL provides an extension to RDF and RDFS. Based on its RDF syntax, ontologies written in OIL are valid RDF documents.

The Defense Advanced Research Projects Agency (DARPA) in conjunction with the W3C is developing the DARPA Agent Markup Language (DAML) by extending RDF with more expressive constructs aimed at facilitating agent interaction on the web [18]. DAML released its first ontology language specification, DAML-ONT in October 2000. In December of the same year DAML+OIL was released to replace DAML-ONT. The formal semantics of DAML+OIL is provided as a mapping to first order predicate logic, written in ANSI Knowledge Interchange Format (KIF) [12]. DARPA maintains an ontology library with near two hundred entries made publicly available at http://www.daml.org/ontologies/. The large adoption and installed base of daml ontologies is making it a favorite language to provide semantic interoperability on the web [3].

More recently, the W3 consortium has released the OWL (web ontology language) as a revision of the DAML+OIL ontology language. OWL is designed to cater the information processing needs of applications, as opposed to human beings. Similarly to DAML+OIL, OWL is intended to represent terms and their relationships in ontological format. OWL has three increasingly-expressive sublanguages: OWL Lite, OWL DL, and OWL Full. OWL Lite supports classification hierarchies and simple constraints, e.g., cardinality. It is intended as quick migration path from taxonomies and thesauri, i.e., that are free from axioms or sophisticated concept relationships. OWL DL supports "maximum expressiveness while retaining computational completeness (all conclusions are guaranteed to be computed) and decidability (all computations will finish in finite time)" [27]. DAML+OIL is equivalent, in terms of expressiveness, to OWL DL. Finally, OWL Full supports maximum expressiveness.
According to the W3 consortium, it is unlikely that any reasoning software will be able to support complete reasoning for every feature of OWL Full. The documentation for OWL was still in the working draft stage by the time this article was written.

Finally there is a proposal for an ontology interchange language, Ontolingua [7]. It was designed to support the design and specification of ontologies using a semantics based on KIF [12]. The set of KIF expressions that Ontolingua allows is defined in an ontology, called the Frame Ontology. The Frame Ontology specifies, in a declarative form, the representational primitives that often supported with special-purpose syntax and code in object centered representation systems. An Ontolingua ontology is composed of classes, relations, functions, objects distinguished and axioms. The expressive power provided by Ontolingua is unmatched by the previous languages surveyed. Unfortunately no reasoning support is provided until this date.

Ontology edition and reasoning support was a key factor in our decision for DAML+OIL as the ontology language used. Another issue that influenced our decision is today’s large community of DAML users and the existence of a public ontology library. We expect to migrate to OWL DL as soon as a more definite documentation is made available by the W3 consortium. We are currently using the OILed tool for edition and the FaCT tool as an inference engine to build our ontologies. OIL’s formal semantics and efficient reasoning support is provided by Description Logics. The semantics of OIL rely on translation into the description logic $SHIQ$ extended with concrete data types, $SHIQ(d)$ . A complete mapping of Oil to $SHIQ(d)$ is available in [20]. OilEd generates DAML extension ontologies, using an export mechanism.

DAML+OIL implements all ontology core elements of the $O$ ontology structure, introduced in the previous section. The terminology mapping is depicted in Table 1. Concepts are mapped to DAML+OIL classes, relations are properties, a class hierarchy is implemented using the subsumption relationship SubClassOf, and the function that relates concepts in a non taxonomic way are mapped to restrictions. The example presented in section 5 is implemented using DAML+OIL.

**Table 1.** Terminology mapping between the $O$ ontology structure and the ontology language DAML+OIL [2].

<table>
<thead>
<tr>
<th>$O$ Ontology Structure</th>
<th>DAML+OIL</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C$ Concept</td>
<td>Class</td>
</tr>
<tr>
<td>$R$ Relation</td>
<td>Property</td>
</tr>
<tr>
<td>$H$ concept hierarchy</td>
<td>Subsumption</td>
</tr>
<tr>
<td>$rel$ function that relates the concepts non taxonomically</td>
<td>Restriction</td>
</tr>
<tr>
<td>$A^O$ Axiom</td>
<td>Axiom</td>
</tr>
</tbody>
</table>

In the next section we present the basis of the proposed ontology development process, the extended lexicon of the language (LEL).
3 Language Extended Lexicon

The Language Extended Lexicon (LEL) is anchored on a very simple idea: *understand the language of the problem without worrying about understanding the problem*. It is a representation of the terms in the application language [23,24]. Application languages, as opposed to domain languages, are very contextualized and represent only a few instances of a given domain. An application language contains conceptualizations shared by a community, where all members of the group understand sentences in the same sense [4]. The main goal of the Lexicon is to capture words or sentences peculiar to the application problem domain.

Each term in the lexicon is represented by two types of description, as opposed to dictionaries and glossaries, that provide a unique description. The first type, *notion*, is the denotation, the intended meaning, of a term or phrase. The notion is equivalent to a description found in a dictionary. The second type of description, called *behavioral response*, is the connotation of the term or phrase, that is, additional meaning provided by the composition with other terms in the lexicon. The behavioral responses describe how a term relates to others in the lexicon, composing new meanings. The Lexicon representation was influenced by the work of Umberto Eco, in which he defines a system of codes based on signs, notions, behavioral responses and a set of rules (the set of relationships that relate signs to notions and behavioral responses) [5]. The terms of the LEL are classified in four categories: object, subject, verb and state. Figure 1 shows an example of a lexicon term of type subject.

![Fig. 1. Example of a lexicon term, substitute².](image)

---

1. The rules are represented in the lexicon by hypertext links.
2. The lexicon term is implemented in the C&L tool, whose interface is in Portuguese. *Nome* translates to name, *Noção* is notion and *Impactos* are the behavioral responses of the lexicon term.
Central to the LEL are two principles. The first is to maximize the use of other lexicon terms when describing the notions and behavioral responses of a new term. This is the principle of closure. The second is to minimize the use of terms, external to the Universe of Discourse. If unavoidable, make sure they belong to the basic vocabulary of the natural language in use, and, as much as possible, have a clear mathematical representation (e.g., set, function). This is called the principle of minimal vocabulary.

Lexicon building is a six-step process. The first step is to identify the main information sources of the Universe of Discourse (UofD). The second step is to identify a list of terms relevant to the UofD using a set of elicitation techniques. The heuristic used in this stage is: list each term that seems to have a special meaning. The next step is to classify the terms. They can be one of object, subject, verb, or state. The terms are described next. In this step the objective is to elicit the meaning of each term. Figure 2 illustrates the process. When describing the terms in the lexicon one should enforce the principles of closure and minimal vocabulary. The lexicon is verified using an inspection strategy [21]. Validation is performed by actors of the UofD using proof reading techniques.

Fig. 2. Lexicon Construction Process [22].

Lexicon edition and maintenance is supported by the open source C&L tool. This tool provides support to the edition and evolution of lexicons and scenarios produced during software development. Lexicons are implemented as hypertext graphs. The tool provides automatic linking to other lexicon terms and corresponding scenarios. The C&L tool was made available as part of the Open Source Project conducted by the Requirements Engineering Group at PUC-Rio. The goal of the project is to provide open source quality software, in Portuguese, to advance the adoption of software practices by small organizations in Brazil. The Portuguese version of the tool can be obtained at http://sl.les.inf.puc-rio.br/cel/aplicacao/. An English version is under way.
4 Lexicon Based Ontology Construction Process

Based on the Language Extended Lexicon, presented on the previous section, we propose an ontology building process. The process is independent of the ontology language used in the implementation. The resulting ontology will be expressed using the core ontology elements defined in section 2, a 5-tuple consisting of concepts, relations, a concept hierarchy, functions that relates the concepts non-taxonomically and axioms \( \{ C, R, \mathcal{H}, \text{rel}, \mathcal{A} \} \) [26]. Figure 3 shows the role of the lexicon in the ontology structure. On the bottom layer we have the LEL, composed of terms classified into verb (\( L^v \)), objects (\( L^o \)), subject (\( L^s \)) and state (\( L^a \)); on the top layer we have the ontology structure \( O \). Process \( P_o \) maps the lexicon terms into ontology elements.

![Ontology Construction Process](image)

Fig. 3. Ontology Construction Process \( P_o \) (inspired in the layered ontology engineering approach proposed by [26]).

We detail process \( P_o \), proposed in [2] as follows:

1. List lexicon terms alphabetically according to their type (verb, object, subject and state).
2. Make 3 lists: concept (C), relations (R) and axioms (\( \mathcal{A}^o \)). In the concept list, each entry will have a name, a description and list of zero, one or more rel (function that relates the present concept to others, non-taxonomically). The entries in the relation and axiom lists will have labels only.
3. Using the list of lexicon terms classified as either object or subject, for each term:
   3.1 Add a new concept to the concept list. The concept name is the lexicon term itself. The concept description is the notion of the term.
      3.1.1 For each behavioral response,
         3.1.1.1 Check the relation list for a relation that expresses it.
         3.1.1.2 If there is none, add new a relation to the relation list.
            The relation name is based on the verb of this behavioral response.
            3.1.1.2.1 Verify consistency
         3.1.1.3 In the concept list, add a new rel to the concept in question. The rel is formed by the concept in question + relation (defined in 3.1.1.1) + concept it relates to
(The concept is the direct/indirect object of the verb in the behavioral response. It is usually a term in the lexicon and appears underlined).

3.1.1.4 Check for negation indicators in the minimal vocabulary that relate the term to other terms. Analyze the pair of terms in order to identify a possible disjoint relationship.

3.1.1.4.1 If true, add the disjoint relationship to the axiom list.

3.2 Verify consistency

4. Using the list of lexicon terms classified as type verb, for each term:

4.1.1 Check the relation list for a relation that expresses it.

4.1.1.1 If there is none, add new a relation to the relation list. The relation name is the lexicon term itself.

4.1.1.1.1 Verify consistency

5. Using the list of lexicon terms classified as type state, for each term:

5.1.1 For each behavioral response

5.1.1.1 Try to identify the importance of the term to the ontology. This strategy is equivalent to the use of competency questions proposed in [15]. The questions could be obtained from rephrasing the behavioral responses of the term in question into questions of type, when, what, who, where, why and how.

5.1.1.2 Check for negation indicators in the minimal vocabulary that relate the term to other terms. Analyze the pair of terms in order to identify a possible disjoint relationship.

5.1.1.2.1 If true, add the disjoint relationship to the axiom list.

5.1.2 If the term seems to be central to the ontology, classify it as a concept (C).

5.1.3 If the term is not central to the ontology, classify it as a relation (R).

5.1.4 Verify consistency

6. When all terms are added to the ontology,

6.1 Check ontology looking for sets of concepts that share identical rel

6.1.1 For each set of concepts that share identical rel, build a separate concept list.

6.1.2 Search the ontology for a concept that refers to all members of this list.

6.1.2.1 If such concept is not found, search the notion and behavioral response of each member of the concept list trying to identify a common term from the minimal vocabulary.

6.1.3 Build a concept hierarchy where every member of the concept list is a sub-concept of the one found in 6.12

6.1.4 Verify consistency
Every time a new element is added to the ontology, a consistency verification is needed. Ideally, the verification task is done with the aid of automated tools. In our practice, as well as in the example we will present in the next section, we use the OilEd tool to edit our ontologies and the FaCT tool, to provide automatic consistency checking.

5 The Meeting Scheduler Ontology: An Example

In this section we exemplify the ontology construction process presented in the last section. We build an ontology for a meeting scheduler. This example was implemented using the OilEd and FaCT ontology using lists of concepts, relations and axioms, as proposed in process P. The FaCT tool provides automated reasoning to support our consistency checks. The ontology was written in the OIL ontology language and exported to the DAML format. The terminology used by OIL is different from the one used by our ontology definition, the O ontology structure proposed by [26]. A complete mapping is provided by Table 1, in section 2. In this section we are going to use OIL terminology. We provide traces to the process P, presented in section 4, using numbers to the process steps in parentheses.

We start by separating the lexicon terms in different listings according to its classification (steps 1 and 2 of process P). The lexicon classification of terms helps distinguishing from lexicon terms that are directly mapped to the ontology from those who need further analysis. Terms marked as object and subject are usually mapped directly into classes (concepts) of the ontology (step 3). Similarly the verb terms are usually mapped into properties (relations) (step 4). Properties are the building blocks of class restrictions (function that relates the concepts non taxonomically – rel) that serve to relate ontology classes (step 3.1.1.3). The state terms have to be evaluated in order to decide whether they would be best modeled as a class or property (step 5). Axioms are derived from an analysis of behavioral responses of terms of types subject, object and state (steps 3.1.1.4 and 5.1.1.2).

We are going to use the lexicon term substitute exemplified in Figure 1 to show the process for a term of type subject (3.1). We interweave the steps of the process with the actions taken in relation to the term substitute, that appears inside light gray shaded boxes. We use font courier to describe the process step used and the actions taken are in bold italics (arial font).

3.1 – Add a new concept to the concept list. The concept name is the lexicon term itself

SUBSTITUTE

3.1 The concept description is the notion of the term

Description (Documentation in OIL):
Person that attends a meeting in the place of another person.
He or she was appointed by the person invited to the meeting.
3.1.1 For each behavioral response, if cannot be present at meeting, sends absence notification.

3.1.1.1 Check the relation list for a relation that expresses it.

Sends – does not exist.

3.1.1.2 If there is none, add new a relation to the relation list. The relation name is based on the verb of this behavioral response.

Create relation (property in OIL) send.

3.1.1.2.1 Verify consistency

Verified using the FaCT tool.

3.1.1.3 In the concept list, add a new rel to the concept in question. The rel is formed by the concept in question + relation (defined in 3.1.1.1) + concept it relates to (The concept is the direct/indirect object of the verb in the behavioral response. It is usually a term in the lexicon and appears underlined).

Create rel (restriction in OIL) “sends absence notification”.

3.1.1.4 Check for negation indicators in the minimal vocabulary that relate the term to other terms. Analyze the pair of terms in order to identify a possible disjoint relationship.

Not found.

3.1.1.4.1 If true, add the disjoint relationship to the axiom list.

3.1.1 For each behavioral response, if can be present at meeting, sends presence confirmation.

3.1.1.1 Check the relation list for a relation that expresses it.

Sends – already exists.

3.1.1.2 If there is none, add new a relation to the relation list. The relation name is based on the verb of this behavioral response.

3.1.1.2.1 Verify consistency

3.1.1.3 In the concept list, add a new rel to the concept in question. The rel is formed by the concept in question + relation (defined in 3.1.1.1) + concept it relates to (The concept is the direct/indirect object of the verb in the behavioral response. It is usually a term in the lexicon and appears underlined).

Create rel (restriction in OIL) “sends presence confirmation”.

3.1.1.4 Check for negation indicators in the minimal vocabulary that relate the term to other terms. Analyze the pair of terms in order to identify a possible disjoint relationship.

Not found.

3.1.1.4.1 If true, add the disjoint relationship to the axiom list.
3.1.1 For each behavioral response, 

3.1.1.1 Check the relation list for a relation that expresses it.

**Approved by – does not exist.**

3.1.1.2 If there is none, add a new relation to the relation list. The relation name is based on the verb of this behavioral response.

**Create relation (property in OIL) approved_by.**

3.1.1.2.1 Verify consistency

**Verified using the FaCT tool.**

3.1.1.3 In the concept list, add a new rel to the concept in question. The rel is formed by the concept in question + relation (defined in 3.1.1.1) + concept it relates to (the concept is the direct/indirect object of the verb in the behavioral response. It is usually a term in the lexicon and appears underlined).

**Create rel (restriction in OIL) “approved_by initiator”.**

3.1.1.4 Check for negation indicators in the minimal vocabulary that relate the term to other terms. Analyze the pair of terms in order to identify a possible disjoint relationship.

**Not found.**

3.1.1.4.1 If true, add the disjoint relationship to the axiom list.

3.2 Verify consistency

**Verified using the FaCT tool.**

To illustrate the example above, we show a screen snapshot of the implementation of the class substitute using the OilEd tool.

Note that the class substitute has a superclass, person#2, appears listed in the classes box – right side, third box from the top of the screen - in Figure 4. The symbol # in the name of the class indicates that the concept person was not defined locally, but borrowed from another ontology. The LEL focus is using the language of the problem to describe the problem. It is based in the principles of closure and minimal vocabulary. Terms entered in the lexicon are only those who bear a special or particular meaning in the Universe of Discourse. Terms used either in the notion or behavioral response of a term but, are not lexicon terms themselves, are part of the minimal vocabulary of the lexicon. Terms that belong to the minimal vocabulary are important, but not special to a particular Universe of Discourse, i.e., their understanding is assumed to be shared among stakeholders thus need not to be entered in the lexicon. This feature is of particular interest to ontology engineering because it makes clear the distinction between terms that have to be implemented (lexicon terms) and terms that can be “borrowed” from existing ontologies (the terms used in the minimal vocabulary).
This is the case of the term *person*. Note that it appears in the notion of the term substitute ("*person that attends a meeting in the place of another person*") see Figure 1. It is not underlined because the term *person* is not a lexicon entry itself, but belongs to the *minimal vocabulary*. The class *person* was added to the meeting scheduler ontology by borrowing from an existing implementation. In this case we used the general.1.0.daml ontology proposed by Jeff Heflin as an ontology that models many general concepts that are required by other ontologies. This ontology is public and available at [11]. The symbol # stands for the alias for namespace #2, the URL where the file containing ontology general.1.0.daml is located. There is no limit to the number of concepts or different ontologies used in the composition of a new one.

Lexicon terms of type verb have to first be checked in the properties list (4.1.1). By the time lexicon terms of type verb are added to the ontology it is possible that its corresponding property was already added as a result of the implementation of the behavioral responses for the terms of types object and subject added in the previous step (3.1.1.2). If that is not the case, we add a new property (4.1.1.1) to the list of existing properties. Our example is the lexicon term *invites*. We show its OilEd implementation in Figure 5. Note that the documentation reflects the very particular meaning (denotation) that the term assumes in the meeting scheduler Universe of Discourse.
A key point in building the ontology is deciding how to implement the state type terms. Lexicon terms of type state can be modeled either as a class or property (5.1.2 / 5.1.3). We recommend the use of ontology competency questions. The latter are the list of questions that a knowledge base based on the ontology should be able to answer [15]. This exercise consists of systematically formulating questions whose answers involve the lexicon term in question. Questions should revolve around determining the nature of the term, how it is to be obtained, what other ontology classes are related to it and what if type of questions. Roughly, the competency questions are the behavioral responses of a lexicon term, reformulated as questions (5.1.1.1).

One of the most important aspects of ontology building is making a clear distinction between the generalization relationships (is-a), structural to the construction of ontologies, from the part-of relationship[16]. Part-of relationships can only be expressed in ontologies by means of properties (3.1.1.2). An example is the class **meeting requirements**. The **meeting requirements** is a document composed of information on the meeting place, meeting materials and the pair meeting date and time. The ontology class **meeting requirements** is bound to other classes by a sequence of restrictions based on the composed_of property (3.1.1.3). Bellow we show the restriction that relates the class **meeting requirements** to the meeting place class using the composed_of property. Because the LEL is built using the principle of closure, it is only natural to suppose that there would be a dual to the restriction “**meeting requirements is_composed_of meeting place**”. This is true and can be found in the class meeting place, restriction “**meeting place is part_of meeting requirements**”, as showed in Table 2.

Consistency verification is done using the FaCT tool. FaCT provides reasoning support by mapping OIL expressions to description logic (S$$^3$$HiQ). The reasoning
services provided in FaCT include finding concept inconsistency and determining subsumption relationships. FaCT is written in Common Lisp. Binaries (executable code) are also available (in addition to the source code) for Linux and Windows systems, allowing FaCT to be used without a locally available Lisp. Moreover, a FaCT server running on a suitable host machine can be used via its CORBA interface on any system that has network access to the server. FaCT is publicly available at [6].

Table 2. Restrictions containing part-of and composed-of properties.

<table>
<thead>
<tr>
<th>class meeting requirements</th>
<th>class meeting place</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>&lt;daml:Restriction&gt;</code></td>
<td><code>&lt;daml:Restriction&gt;</code></td>
</tr>
<tr>
<td><code>&lt;daml:hasClass&gt;</code></td>
<td><code>&lt;daml:hasClass&gt;</code></td>
</tr>
<tr>
<td><code>&lt;daml:Class rdf:about=&quot;file:C:/OilEd/ontologies/meeting_v3.daml#meeting_requirements&quot;/&gt;</code></td>
<td><code>&lt;daml:Class rdf:about=&quot;file:C:/OilEd/ontologies/meeting_v3.daml#meeting_requirements&quot;/&gt;</code></td>
</tr>
<tr>
<td></td>
<td><code>{/daml:hasClass}</code></td>
</tr>
<tr>
<td><code>{/daml:Restriction}</code></td>
<td><code>{/daml:Restriction}</code></td>
</tr>
</tbody>
</table>

6 Conclusion

In this paper we proposed a lexicon based ontology development process to support non experts develop machine processable ontologies in a systematic way. The lexicon provides support to the elicitation, modeling and analysis of both concepts and relationships relevant to the Universe of Discourse of the application. Because it based on well defined construction process, the lexicon has been successfully used by experts and non experts alike. Lexicon terms are captured using semi-structured natural language. Differently from usual dictionaries, that only capture the meaning of terms (connotation), the LEL also registers the impacts of the term in relation to other terms in the lexicon (denotation). In addition, all lexicon terms are classified in pre defined categories. This structure provides a starting point to the ontology modeling of concepts, for some of the lexicon term categories have a direct mapping to ontology elements, e.g., lexicon entries of type object are directly mapped to ontology classes.

Our contribution, in addition to providing a systematic ontology construction process, is providing a solution to the problem of ontology scoping. Limiting the scope of new ontologies, i.e., deciding what terms to implement and which ones to borrow from other ontologies is known to be a hard design decision [8]. Our approach contributes in providing a clear separation between new terms to be implemented by the ontology and terms that could be “borrowed” from existing ontology libraries.

The proposed process is currently automated by the ontology plug-in prototype of the C&L tool. Requesting user intervention at specific times, the tool is able to guide the transformation process of a lexicon, captured by the C&L tool, into an ontology. The output of the plug-in is a DAML file, that can be viewed using any ontology tool. We did not, however, implement a verification engine. For that purpose, we generate the DAML file and use FaCT to verify ontology consistency. Early experimentation with the tool has showed that the use of automation, although user intervention is still required, made the process much speedier and self explanatory. Previous experiments
with the manual application of the process have generated a number of (non expert) user questions and doubts about the process that were reduced by less than half, when the tool assisted process took place. Future research includes the implementation of an Open Source verification engine.

Our future work includes continuing experimentation with the proposed process. We are currently processing information from two case studies done in the area of food services. Preliminary results show that the approach is feasible and contributes to a larger understanding of the problem domain. We identified a need for further automated validation techniques and will concentrate future efforts in this area. We are going to investigate how software agents can provide feedback on the quality of the ontologies they are using. Most ontology libraries available in the web admit new entries with little or no reviewing process, which naturally raises questions to the quality of the available sources. In order to promote ontology integration and the use of existing ontologies, it is paramount to have a means of measuring the quality of the sources used.

References

11. Heflin, J. - heflin@cs.umd.edu - An ontology that models many general concepts that are required by other ontologies. This is the DAML version of a SHOE ontology.


Multi-agent Systems
and Security Requirements Analysis

Paolo Bresciani¹, Paolo Giorgini², Haralambos Mouratidis³, and Gordon Manson⁴

¹ ITC-irst, via sommarive 18, I-38050, Povo, Trento, Italy
bresciani@itc.it
² Department of Information and Communication Technology, University of Trento, Italy
paolo.Giorgini@dit.unitn.it
³ School of Computing and Technology, University of East London, England
haris@uel.ac.uk
⁴ Department of Computer Science, University of Sheffield, England
g.manson@dcs.shef.ac.uk

Abstract. Agent Oriented Software Engineering (AOSE) is a software paradigm that has grasped the attention of researchers the last few years. As a result, many different methods have been introduced to help developers in the development of multi-agent systems. However, so far, these methods have mainly neglected security requirements. The common approach towards the inclusion of security within a system is to identify security requirements after the definition of the system. However, this approach has provoked the emergence of computer systems afflicted with security vulnerabilities. In this paper we propose an analysis, based on the measures of criticality (how critical an actor of the system is) and complexity (represents the effort required by the actors of the system to achieve the requirements that have been imposed to them), which aims to identify possible bottlenecks of a multi-agent system with respect to security. An integrated agent-based health and social care information system is used as a case study throughout this paper.

1 Introduction

In a world that becomes more and more reliant on software systems, security is an important concern [1,2]. Private information is stored in computer systems and without security organizations are not willing to share information or even use the technology. In addition, possible security breaches can cost huge amount of time and money.

Following the wide recognition of multi-agent systems, agent-oriented software engineering has been introduced as a major field of research. Many agent-oriented software engineering methodologies have been proposed [3,4,5] each one of those offering different approaches in modeling multi-agent systems. However, only few attempts [6,7] have been made to integrate security issues within the development stages of methodologies.

One of the reasons is the fact that security requirements are generally difficult to analyse and model. It is difficult to analyse because many times security requirements conflict with functional requirements and many trade offs are required. Performing
such trade-offs can be painful and time-consuming and it requires software and security engineering expertise. A second reason is lack of developers’ acceptance and expertise for secure software development [8].

When considering security issues during the development of multiagent systems, the goal will be to provide as much security as possible trading sometimes security concerns with other functional and non-functional requirements. To better achieve this goal, agent-oriented software engineering methodologies must help developers, through a systematic approach, to determine how complex is for each part (actor) of the system to achieve the security requirements, and also identify the most critical actors of the system with respect to security. Such an approach will help developers to perform trade-offs between security and other functional and non-functional requirements based on quantitative measurements and thus minimizing the risks of putting in danger the security of the system.

Within a multi-agent system, more likely, different agents will play different roles and, with respect to security, some will be more critical than others. In addition, some agents of the system might have been overloaded (assigned more security requirements than they can handle) and thus fail to satisfy some of the security requirements assigned to them.

Developers must be able to identify, through a systematic approach and without much security knowledge, such cases and redefine the design of the system in such a way that the agents of the system are not overloaded and all the security requirements assigned to the agents of the system are satisfied.

In this paper we propose an approach based on the concepts of criticality and complexity, and we indicate how this approach can be integrated within the early requirements analysis stage of the Tropos methodology. This work is within the context of the Tropos project [4] and our aim is to provide a clear and well-guided process of integrating security, functional and non-functional requirements throughout the whole range of the development process. Section 2 provides an overview of Tropos methodology, and also introduces the electronic Single Assessment Process (eSAP) system case study. In Section 3, we describe the process of analysing the complexity and criticality of a system with respect to security, and we present an algorithm to reduce the complexity and/or the criticality of the overloaded actors. Finally, Section 4 presents some concluding remarks and directions for future work.

2 Tropos Methodology

Before we can describe our approach, we think it is necessary to provide an overview of Tropos methodology and how security can be integrated to it. Tropos is an agent oriented software engineering methodology, tailored to describe both the organisational environment of a system and the system itself, employing the same concepts throughout the development stages. The Tropos methodology is intended to support all the analysis and design activities in the software development process, from the application domain analysis down to the system implementation [4]. Using Tropos, developers build a model of the system-to-be and its environment that is incrementally refined. One of the key features of the methodology is the use of the notion of agent and related mentalistic notions, which are founded on BDI (Belief, Desire, and Intention) agent architectures [9], in all software development phases, from early requirements analysis down to the actual implementation.
Tropos adopts Yu’s i* model [10] which offers the concepts of actors, goals, tasks, resources and social dependencies for defining the obligations of actors (dependees) to other actors (dependers). Actors have strategic goals and intentions within the system or the organisation and represent (social) agents (organisational, human or software), roles or positions (represent a set of roles). A goal represents the strategic interests of an actor. In Tropos we differentiate between hard goals (only goals hereafter) and soft goals; the latter having no clear definition or criteria for deciding whether they are satisfied or not. A task represents a way of doing something. For example a task can be executed in order to satisfy a goal. A resource represents a physical or an informational entity while a dependency between two actors indicates that one actor depends on another to accomplish a goal, execute a task, or deliver a resource.

Tropos covers four stages of software development:

**Early Requirements** analysis that consists of identifying and analysing the stakeholders and their intentions. Stakeholders are modeled as social actors, while their intentions are modeled as goals that, through a goal-oriented analysis, are decomposed into finer goals, which eventually can support evaluation of alternatives.

**Late Requirements** analysis that consists of analyzing the system-to-be within its operating environment, along with relevant functions and qualities. The system is introduced as an actor and the dependencies between the system and the other actors of the organization are explicitly modeled. These dependencies define the system’s requirements.

**Architectural Design** that describes the system’s global architecture in terms of subsystems (actors) interconnected through data and control flows (dependencies). During this stage, new actors are introduced in the system as a result of analysis performed at different levels of abstraction. In addition, capabilities needed by the actors to fulfill their goals and tasks are identified.

**Detailed Design** that deals with the specification of each architectural component in terms of inputs, outputs, control and other relevant information. Tropos faces the detailed design stage on the basis of the specifications resulting from the architectural design stage and the reasons for a given element can be traced back to the early requirements analysis.

The security process in Tropos consists of analyzing the security needs of the stakeholders and the system in terms of security constraints [11] imposed to the stakeholders (early requirements) and the system (late requirements), identifying secure entities [11] that guarantee the satisfaction of the security constraints, and assigning capabilities to the system (architectural design) to help towards the satisfaction of the secure entities.

In our work [6,11] we define security constraints as constraints that are related to the security of the system whereas secure entities represent any secure goal/task/resource of the system [11]. Security constraints can be categorized into Positive – they influence the security of the system positively (e.g., *Allow Access only to Personal Information*) – or negative – they influence the security of the system negatively (e.g., *Send information plain text*).

To make the process easier to understand, we consider as an example the electronic Single Assessment Process (eSAP) case study first introduced by Mouratidis et. al [12]. The eSAP case study involves the development of an agent-based health and
social care system for the effective care of older people. Security in such a system, as in any health and social care information system, is very important since revealing a medical history could have serious consequences for particular individuals. It is worth mentioning that in this example we only focus on confidentiality, but our approach can be also applied for integrity and availability.

Taking into account a substantial part of the eSAP, we have defined the following stakeholders for our case study: The Older Person (OP) actor is the older person (patient) that wishes to receive appropriate health and social care. The Professional actor represents health and/or social care professionals involved in the care of the Older Person. The DoH actor represents the English Department of Health, which is responsible for the effective care of the Older Person. The Benefits Agency actor is an agency that helps the Older Person financially, and the R&D Agency represents a research and development agency interested in obtaining medical information.

During the early requirements analysis stage, the dependencies, the goals and the security constraints between these actors can be modeled using Tropos actors’ diagram as shown in Figure 11.

In such a diagram each node represents an actor, and the links between the different actors indicate that one depends on another to accomplish some goals. It is also worth noticing that the security constraint is placed close to the corresponding goal or soft goal on the side of the actor who requires the constraint.

In our example, the Older Person depends on the Benefits Agency to Receive Financial Support. However, the Older Person worries about the privacy of their finances so they impose a constraint to the Benefits Agency actor, to keep their financial information private. The Professional depends on the Older Person to Obtain Information, however one of the most important and delicate matters for a patient (in our case the Older Person) is the privacy of their personal medical information, and the sharing of it. Thus most of the times the Professional is imposed a constraint to share this information if and only if consent is achieved. One of the main goals of the R&D Agency is to Obtain Clinical Information in order to perform tests and research. To get this information the R&D Agency depends on the Professional. However, the Professional actor is imposed a constraint (by the Department of Health) to Keep Patient Anonymity.

3 Criticality and Complexity

In the previous section we have briefly described a process of analysing the security of an organisational setting taking into consideration some security constraints, which are imposed by the different stakeholders. However, it is more likely to find different security constraints that have different impact on the security of the system. That is one constraint might endanger the security of the system in a level that must be satisfied even if it involves a trade off with some other functional or non-functional requirements. On the other hand, some constraints might not be as important. As a result, different actors of the system impact the security of the system differently according to what security constraints have been imposed to.

---

The numbers next to the security constraints represent the criticality of the constraint (Section 3).
3.1 Security Criticality

It is important to provide an analysis that identifies the impact each actor has on the security of the system. In doing so we need to define how critical each security constraint is for the overall security of the system. We call this measure, security criticality and we define it as follows:

**Security Criticality** is the measure of how the security of the system will be affected if the security constraint is not achieved.

Security criticality allows us to evaluate how critical each actor of the system is with respect to security. This will help us to identify the security bottlenecks of the system.

---

2 Criticality has been introduced by E. Yu in [4].
system, and refine it by taking into consideration the different impact that each actor has on the security of the system. We differentiate between \textit{ingoing} and \textit{outgoing} security criticality. Ingoing security criticality is the security criticality that actors assume when they are responsible for achieving a security constraint. On the other hand, the outgoing security criticality represents the security criticality of the achievement of a constraint for the actor that imposes the security constraint.

In order to calculate the criticality of the system, we consider the dependencies and we assign a value for each security constraint (see numbers next to security constraints in Figure 1). These values were assigned after closely studying the system’s environment and after discussing them with the stakeholders. In the case of an open secure dependency (a dependency that has no security constraints attached to it), we assign a value of zero both for the ingoing and outgoing criticalities.

In this example we have assumed that criticality obtains integer values within the range 1-5, where 1 = very low, 2 = low, 3 = medium, 4 = high, 5 = very high. However, the range of acceptable values can change and it depends on each developer. For example, developers might decide it is better for them to assign values within the range 1-20. This will provide them with more accurate ratings of the criticalities.

In addition, a maximum value of criticality is defined for each actor taking into account, the actor’s abilities, their available time, and the responsibilities they have in the organization.

3.2 Security Complexity

Security criticality allows us to evaluate how critical each actor of the system is with respect to security. Nevertheless, we need to be able to evaluate how much effort is required by each of the actors to achieve their security constraints. To perform such an evaluation, we introduce the concept of security complexity and we define it as follows:

\textbf{Security Complexity} is the measure of the effort required by the responsible actor for achieving a security constraint.

Security complexity helps to design sub-systems to support actors that might be in danger of not achieving some security constraints that have been imposed to them, and as a result endanger the overall security of the system. This means, if an actor is overloaded with security responsibilities, some of the security constraints should be delegated to another existing actor of the system, or if this cannot happen, the developer should introduce another actor and delegate some of the security constraints of the overloaded actor.

In order to be realistic, we need to take into account both the system and security complexity, where \textbf{System Complexity} is defined as the measure of the effort required from the dependee for achieving the dependum [13]. This is necessary since it might be the case that an actor’s security complexity is high, however since their system complexity is very low, they are capable of achieving all the security constraints. On the other hand, there might be cases where an actor’s security complexity might be low but their system complexity is high and therefore they might not be able to achieve all the security constraints imposed to them. Thus, by taking into consideration both system and security complexity we can identify more precise the degree of achievement of the security complexity.
Moreover, an important factor in (realistically) calculating the overall complexity is time. It might be the case that an actor can achieve different (secure) goals sequentially, so in this case it would not be realistic to sum up the individual values of complexity in order to evaluate the overall complexity of the actor. Sum up all the different complexity values would be realistic only if all the goals should be achieved at the same time. However, in the real world this will be more likely the case of an organization (department) in which different agents work, than the case of a single agent.

Similar to criticality analysis, we have assumed that complexity (system and security) can obtain integer values within the range 1-5, where 1 = very low, 2= low, 3=medium, 4=high, 5=very high. Also similarly to criticality, a maximum value of (overall) complexity is defined for each actor.

To be able to precisely assign values for security and system complexity, each actor of the system and their security constraints and goals respectively must be further analysed. This is necessary because the security constraints and the goals modeled in the actors’ diagram (figure 1) are quite superficial and it is difficult to evaluate their complexity. Therefore, many different alternative tasks might be considered for their satisfaction, each with different complexity value. To cope with this, we are extending our analysis, by further analysing (for each actor involved in our system) the security constraints (for the security complexity) and the actor’s goals (system complexity), together with the different alternatives that can satisfy them. This kind of analysis, apart from helping us to define more precisely the values for complexity, it provides a basis to choose between different alternatives that can be employed for the satisfaction of security constraints and the actor’s goals, something very important in justifying the trade offs between security and the functional requirements of the system.

To help us in this analysis, we are employing Tropos rationale diagrams [10]. Differently than actors’ diagram, which focuses on the external relationships between the actors of the organization, each rationale diagram analyses the internal goals, security constraints and dependencies of each actor (figure 2). In order to calculate the values of security complexity for each actor, different weights have been assigned to the different relationships involved in the satisfaction of the security constraints (secure goals), that have been imposed to the actor, and the actor’s strategic goals. For reasons of simplicity in this paper we have assumed weights can obtain integer numbers in the range of 1-5 (1 being the lowest value with respect to complexity and 5 the highest).

In addition, in the cases where the dependum is a soft goal, minimal system complexity values are assumed. This is the minimal effort requested from the depender to achieve the soft goal. This has been decided since the concept of a soft goal has no clear criteria for whether there are satisfied or not, and as such we cannot assign a precise value required for achieving the soft goal.

### 3.3 An Example

Going back to our case study, the rationale diagram of the **Professional** actor is shown in figure 2. As it can be seen from the figure, different alternatives can be considered for the satisfaction of the security goals imposed to the actor as well as the actor’s strategic goals. For example, the **identify problems** goal of the **Professional** actor can be satisfied either by evaluating the information manually (**Evaluate Medical Info Manually** task) or by using the eSAP system (**use eSAP** task). For each of those alternatives we have assigned a value as shown in figure 2.
In addition, the contribution of each alternative to the other functional and security requirements is shown in figure 2 (as dashed line links). To denote the contributions of the different alternatives, we employ a quantitative approach presented by Giorgini et al [14]. Each contribution receives weights between 0 and 1, where 0 means the alternative puts in maximum danger the security or the functional requirement, whereas 1 means the alternative completely satisfies the security or the functional requirement. To keep the diagram simple and understandable we denote contributions to the Keep Patient Anonymity security constraint, resulting only from the Obtain OP Consent secure goal alternatives (figure 2).

For example, the Share Information Only if Consents Obtained security constraint of the Professional actor is satisfied by the Obtain OP Consent secure goal. However, this goal can be achieved by considering different alternatives, each one of those alternatives having a different security complexity weight. Thus, the Professional can Visit OP, Use Phone, Use eSAP, or Ask a Nurse to obtain the consent of the Older Person. These tasks have been assigned with different weights of complexity according to how much effort is required from the Professional to achieve them. Thus, in the above-mentioned tasks we have assign weights of 5, 4, 3 and 2 respectively. However, in deciding which task is best suited, developers should also consider how this task affects (if it affects) other requirements of the system. Consider, for example, the Ask a Nurse task of the Professional. Although this is the simplest task and it could have been the obvious choice from the point of view of complexity, it is worth considering that the involvement of a nurse could contribute negatively to the Keep Patient Anonymity security constraint also imposed to the Professional actor. This could endanger the privacy of the Older Person, an undesired effect for our system. Thus, in this case, we have decided to choose the Use eSAP task, since it requires the less effort (apart from the Ask a Nurse) and also it helps towards the older person’s privacy. When all the different options have been considered and a choice about which one is best suited have been made, the next step is to calculate the overall
complexity for each actor. This process takes part alongside with the calculation of the criticality for each actor.

The first step in analyzing the complexity and criticality, with respect to security, involves the calculation of their values for each actor involved in the system. Then, if some actors are overloaded, i.e. they assume a greater value of complexity and criticality than the maximum value they can handle, some of the security constraints related to these actors have to be reassigned to different actors of the system in order to reduce the complexity or the criticality of the overloaded actors. Therefore, the problem we want to solve is summarized as follows:

_How to reassign one (or more) goals of actors whose complexity/criticality is greater than their maximum complexity/criticality limit?

In other words, how can we reconfigure the topology of the actor diagram in order to end up with a “balanced” configuration? Of course we would like to solve the problem by means of minimal topology modifications. In fact, many solutions may be found by radically redesigning the diagram, but these shouldn’t be considered as first choice solutions.

To take into account these needs, we propose in Figure 3 the Rebalance algorithm that, given a representation of an actor diagram and its constraints, is capable to produce a new configuration (if it exists), in which the constraints are satisfied. For the sake of simplicity, the presented algorithm considers only the complexity and not the criticality. However, it is relatively easy to extend the algorithm to consider both the complexity and the criticality.

Let us assume there are \( m \) dependums and \( n \) actors. Moreover, let us suppose that the fact that different actors may fulfill the different dependums, is coded by means of a cost matrix \( \text{CoM}[1..n, 1..m] \) where, for each actor \( i \) and dependum \( j \), the cost for \( i \) to fulfill \( j \) is \( \text{CoM}[i, j] \). This cost may be different for different actors fulfilling a given dependum (not all the actors have the same level of skills) and, in particular, it may be infinite (MAXINT) for some actors (not all the actors can fulfill a given dependum). On the other hand, the vector \( \text{M_CoV}[1..n] \) provides the maximum complexity that each actor can hold. It is worth mentioning that the matrix \( \text{CoM} \) and the vector \( \text{M_CoV} \) are constant data provided with the analysis of the domain.

In addition, the actor diagram topology, is described by means of a variable \( \text{A}[1..n, 1..m] \) of booleans where a “1” in position \((i, j)\) means that the dependum \( j \) is assigned at the actor \( i \). Of course, for each dependum \( j \) there is one and only one “1”. The actor load defined by the current topology is computed by the Function:

\[
\text{Compl}(i, \text{A}) = \sum_{j=1}^{m} \text{CoM}[i, j]\text{A}[i, j]
\]

The core of the algorithm is given by the Function _Try_One_Actor_ (Figure 4) that tries to rearrange the matrix \( \text{A} \) in order to accommodate the load of actor \( i \) below its maximum complexity capacity, starting to analyze dependum \( j \) first. It iteratively considers possible reassignments for dependum \( j \) to other actors that can fulfill it without exceeding their maximum capacity. This possibility is tested by the Function:

\[
\text{Fits}(\text{A}, l, j) = (\text{Compl}(l, \text{A}) + \text{CoM}[l, j] \leq \text{M_CoV}[l])
\]
CONST m:integer; {# of dependums}
n:integer; {# of actors}
M_CoV:array[1..n] of real;
{max cost for each actor}
CoM: array[1..n,1..m] od real;
{the effort for actor i to provide goal j}

GLOBAL VAR VISITED_DEP: set of visited dependums;
{initially empty}
A: array[1..n,1..m] of boolean;
{the assignment matrix properly
initialized to reflect diagram
topology}

LOCAL VAR SET_OF_UNBALLANCED: set of actors;

Function Rebalance(var A: ass_matrix): boolean;
begin
result:=Reballance_Intransitive(A);
if result=fail then
begin
SET_OF_UNBALLANCED:={i|Compl(i,A)>M_CoV[i]};
copy_of_A:=A;
while result=fail and
not empty(SET_OF_UNBALLANCED) do
begin
i:=POP(SET_OF_UNBALLANCED);
result:=Try_Transitive(i,A);
if result=fail then A:=copy_of_A
end
end;
RETURN result
end;

Fig. 3. The reassignment algorithm

The problem is recursively scaled down by considering also other dependums (j+1) if the reassignment of the current one (j) is not sufficient or not possible. Backtrack is required in case the current reassignment of j to l is useless.

The above presented core function, considers only one overloaded actor. However, it can be extended to consider more overloaded actors (see the Function Reballance_Intransitive in Figure 5). Such function is recursively called (with possible backtrack) only in the case at least one of the overloaded actors can be re-balanced. Backtrack allows us to iteratively consider all the overloaded actors in turn as the first to be processed. In fact, the solution may depend, in a generic —even tough very idiosyncratic— case, by the processing order. The recursion takes care for considering the other overloaded actors.

Finally, if a solution involving the redistribution of dependums from actor to actor requiring that recipient actors have not to be re-balanced themselves cannot be found by means of the Function Reballance_Intransitive (Figure 5), the more generic and entry point Function Rebalance try to consider also the possibility of transitively affect the load of recipient actors even over their maximum capacity, by calling the Function Try_Transitive (Figure 4). In this case the adjustments can
Function Try_One_Actor(i,j: integer; var A: ass_matrix): boolean;
begin
  if j>m then RETURN fail;
  result:=fail;
  l:=0;
  if A[i,j]=1 then
    while Compl(A,i)>M_CoV[i] and l<n do
      begin
        l++; 
        if l<>i and Fits(A,l,j) then
          begin
            copy_of_A:=A;
            A[l,j]:=1; A[i,j]:=0;
            if Compl(A,i)>M_CoV[i] then
              begin
                result:=Try_One_Actor(i,j+1,A);
                if result=fail then A:=copy_of_A
              end
            else result:=OK
          end
        end;
      end;
    if result=OK then RETURN result
    else RETURN Try_One_Actor(i,j+1,A)
  end;
Function Try_Transitive(i: integer; var A: ass_matrix): boolean;
begin
  result:=fail;
  copy_of_A:=A;
  j:=0;
  if not empty(VISITED_DEP) then
    while j<m and result=fail do
      begin
        j++;
        if A[i,j]=1 and not j in VISITED_DEP then
          begin
            push(j,VISITED_DEP);
            l:=0;
            while l<n and
              (Compl(i,A)>M_CoV[i] or result=fail) do
              begin
                l++;
                if l<i and CoM[l,j]<MAXINT then
                  begin
                    A[l,j]:=1; A[i,j]:=0;
                    result:=Reballance(A);
                    if result=fail then A:=copy_of_A
                  end
              end;
          end;
        if result=fail then VISITED_DEP:=VISITED DEP-{j}
      end;
  RETURN result
End

Fig. 4. The functions Try_One_Actor and Try_Transitive
be spread all over the matrix, implying radical topology redesign. Minimizing modifications became now more difficult even to be defined, and, in the current version of the algorithm, no particular claim is done, except that termination and the production of one solution (if it exists) is guaranteed. Termination is guaranteed by the fact that each dependum is reassigned at most once (there is no need to reassign it more than once; again, of course, the use of backtracking allow us to test all the re-assignments).

```
Function Reballance_Intransitive (var A:ass_matrix):boolean;
begin
  result:=fail;
  SET_OF_UNBALLANCED:={i|Compl(i,A)>M_CoV[i]};
  if empty(SET_OF_UNBALLANCED) then result:=OK
  else
    begin
      copy_of_A:=A;
      while result=fail and not empty(SET_OF_UNBALLANCED) do
        begin
          i:=POP(SET_OF_UNBALLANCED);
          if Try_One_Actor(i,1,A)=OK then
            begin
              result:=Reballance_Intransitive(A);
              if result=fail then A:=copy_of_A
            end
        end
    end;
  RETURN result
end;
```

Fig. 5. The function Reballance_Intransitive

4 Conclusions

In this paper we have presented an analysis for evaluating the degree of complexity and criticality of the actors of the system, with respect to security. Such an analysis provides a valuable process for the developers of multi-agent systems in order to identify possible security bottlenecks. In addition, we have proposed an algorithm to reduce the complexity or the criticality of the “overloaded” actors.

Our analysis helps to justify possible trade-offs between security and functional requirements. By knowing how critical an agent is with respect to security a decision can be made. Our aim is to provide a clear well guided process of integrating security and functional requirements throughout the whole range of the development stages. Such a process must use the same concepts and notations throughout the development phases. The ability to identify the bottlenecks of a multi-agent system with respect to security and justify the decisions behind possible trade-offs between security and functional requirements can definitely help towards this aim.

It is worth mentioning that in this paper we only consider security requirements. Nevertheless, our approach can be easily adapted to deal with other non-functional requirements.

This work is an ongoing research. The presented analysis covers only the requirements stage of the Tropos methodology. We are working towards extending our
analysis to the next stages of the methodology, since such an analysis can help in the later stages of the development. For example, criticality and complexity can help us to decide for different architectural choices during the architectural design stage of the methodology, such as the choice between mobile and static agents.

In addition, we are working towards the development of a process that will allow developers to assign weights to different alternatives in case the different stakeholders disagree on the assignment.

The present version of the algorithm guarantees to find a solution that requires the reassignments of dependums of overloaded actors only, if it exists. Otherwise, a solution with transitive reassignments is in any case provided (if it exists), although we cannot at present guarantee it is the best. We believe that, possibly after small improvements, the algorithm can provide the “best” solution. We foresee to work to prove this fact. Moreover, our future research plan includes also the study of the complexity of the algorithm, and its implementation and test.

Acknowledgements

The third Author is grateful to the RANK Foundation for the funding of his research project, in which this work was carried out.

References


Separation of Concerns in Multi-agent Systems: An Empirical Study

Alessandro Garcia¹, Cláudio Sant’Anna¹, Christina Chavez², Viviane Torres da Silva¹, Carlos J.P. de Lucena¹, and Arndt von Staa¹

¹ PUC-Rio, Computer Science Department, LES, SoC+Agents Group
Rua Marques de São Vicente, 225 - 22453-900, Rio de Janeiro, RJ, Brazil
{afgarcia,claudios,viviane,lucena,arndt}@inf.puc-rio.br
http://www.teccomm.les.inf.puc-rio.br/socagents
² UFBA, Computer Science Department
Av. Adhemar de Barros, s/n – 40170-110, Salvador, BA, Brazil
flach@ufba.br

Abstract. With multi-agent systems (MASs) growing in size and complexity, the separation of their concerns throughout the different development phases is crucial to MAS engineers. Separation of concerns is a well-known principle in software engineering to achieve improved reusability and maintainability of complex software. Hence it is necessary to investigate systematically whether abstractions from object-oriented (OO) software engineering are able to isolate explicitly MAS concerns. This paper presents an empirical study that evaluates the degree to which abstractions associated with two OO techniques enable modularization of MAS concerns. The selected techniques involve basic OO abstractions, such as classes and objects, and more sophisticated ones, namely design patterns and aspects. The gathered results shown that the use of aspects allowed the construction of a MAS with a significant improvement in the modularization of different concerns. Also, the use of aspects resulted in: (i) fewer lines of code, (ii) fewer design and implementation components, and (iii) lower coupling between the components. However, the aspect-oriented approach produced lower cohesion in the MAS components. Finally, an important finding of this empirical study is that aspects supported a better alignment with higher-level abstractions from agent-oriented design models.

1 Introduction

Software engineering of large multi-agent systems (MASs) involves a number of concerns, such as agent types, autonomy, adaptation, collaboration, and roles. Many MAS concerns are different from classical concerns addressed in object-oriented (OO) software engineering. However, developers currently rely the handling of these MAS concerns on OO design techniques [26] and programming languages, such as Java. The production of maintainable and reusable MASs requires a deep understanding of the interplay between MAS concerns and OO abstractions. The reuse and maintenance of these MASs depend largely on the ability of such abstractions to support the explicit separation of MAS concerns throughout the design and implementation stages.
So far, few systematic studies have been conducted to investigate the interplay between agents and OO software engineering. A number of OO implementation frameworks have been proposed for MAS development. However, no empirical study reported in the literature tries to improve our understanding of how OO abstractions impact the separation of recurring concerns in MAS development (Fig. 1). In addition, it is not clear to which extent widely-used and evolving abstractions enforce the observance of other desirable principles in MAS engineering, such as low coupling and high cohesion.

Research on agent-based software engineering has primarily focused on the development of agent-oriented methodologies and modeling languages, without focusing on the relationships between agents and objects [18, 23]. Many researchers [23, 24] argue persuasively that the concerns associated with MASs are often much different from those traditionally associated with OO systems; and hence the OO abstractions generally fail to capture the relevant concerns of MASs. However, this statement originates from experts’ opinions and observations based on informal experience but are not a result of empirical evidence. Experimental studies [4] are the most effective way to supply empirical evidence that may improve our understanding about software engineering phenomena.

In this context, this paper presents the results of an experiment that makes systematic use of two different OO techniques for MAS development. The overall goal of this study is to evaluate the maintenance and reuse support of the investigated techniques for addressing MAS concerns. These techniques are associated with basic OO abstractions, such as classes and objects, and more sophisticated ones, namely design patterns and aspects. Patterns and aspects are not restricted to the object paradigm, but in this work we focus the use of these abstractions on the OO context. Both aspect-
based and pattern-based techniques are used in our work to develop two versions of a reactive MAS that supports virtual development of web portals.

The evaluation of the two systems is conducted by using a proposed quality model and Basili’s GQM methodology [3] based on which our metrics suite is identified. The software metrics were applied to provide software engineers with a better understanding of the interplay between the OO abstractions and the agency concerns. The metrics are based on important software engineering properties, including separation of concerns, coupling, and cohesion. Moreover, several scenarios generated during the case study are used to evaluate the reusability and maintainability of MASs.

The results gathered in our study provide a clear understanding of the strengths and weaknesses of the two investigated techniques and their compatibility and divergences. The results are important sources towards the potential convergence of OO and agent-oriented techniques. They are also useful for engineers of realistic MASs who need to implement the agent-oriented models using OO programming languages. The conclusions may also be of interest to agent-oriented methodologists since they may decide to incorporate solutions for problems detected in our study directly as part of their methodologies.

The remainder of this paper is organized as follows. Section 2 presents the investigated techniques and the associated methods. Section 3 discusses the organization of the study in terms of the methodology used, associated goals and questions, stated hypotheses, subjects involved and the used project. Section 4 presents our evaluation framework, which is composed of a quality model, related metrics and a set of scenarios. Section 5 presents the study results, which are interpreted and discussed in Section 6 based on the stated hypotheses. Section 7 includes some concluding remarks and directions for future work.

2 The Investigated Techniques

In our experimental study, we have selected two techniques for MAS development: pattern-oriented development [7,15] and aspect-oriented development [28,47]. The reason why these techniques have been chosen is because they encompass a wide range of basic OO abstractions and additional ones, namely patterns and aspects. Furthermore, these additional abstractions are regarded as promising solutions to improve separation of concerns, reduce coupling, increase cohesion, and promote software reuse and maintenance.

Each technique uses a different set of abstractions that are illustrated in Table 1. The abstractions are classified into: components, component elements, and relationships. Note that the set of abstractions used by each technique is not disjoint. Both techniques explore the classical abstractions from OO software engineering such as classes, objects, inheritance, association, and so on. The basic difference is that the first one is centered on the use of the pattern abstraction and the second one is centered on the use of the aspect abstraction.

Since the investigated techniques were not developed with MAS concerns (or agency concerns) in mind, two supporting methods, specially tailored to the MAS development, were associated with each technique. The experimental subjects used the methods to assist the application of the respective technique and associated ab-
A pattern-oriented method [21] was used to apply existing patterns [7, 15, 26] to the MAS context. Similarly, an aspect-oriented method [20] was used to guide the use of aspects to deal with MAS concerns. The purpose of both methods is threefold: (1) to promote separation of agency concerns based on abstractions of the respective technique, (2) to support the construction of reusable and maintainable large MAS, and (3) to minimize the misalignments between high-level agent-based models and OO design and implementation.

Table 1. The abstractions associated with the investigated techniques

<table>
<thead>
<tr>
<th>ABSTRACTIONS</th>
<th>Central Abstractions</th>
<th>Components</th>
<th>Component Elements</th>
<th>Relationships</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pattern-Oriented Technique</td>
<td>Patterns</td>
<td>Classes, Objects, Interfaces, Abstract Classes</td>
<td>Attributes, Methods</td>
<td>Inheritance, Association, Aggregation, …</td>
</tr>
<tr>
<td></td>
<td>Classes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Aspect-Oriented Technique</td>
<td>Aspects</td>
<td>Aspects, Aspect Instances, Abstract Aspects</td>
<td>Advices, Join points</td>
<td>Crosscutting</td>
</tr>
</tbody>
</table>

Each method step achieves separation of MAS concerns through the isolation of the concerns in individual abstractions or through a set of interrelated abstractions. The methods minimize the misalignments and cover gaps between abstractions from agent-based modeling and OO design by indicating which OO abstraction(s) to use for given abstractions from high-level agent models. Both methods are independent of MAS implementation frameworks, such as JADE [5], ZEUS [35], and Retsina [46], and specific communication languages, such as ACL [14] and KQML [13]. This independence is important for the scope of this work since we are not concerned with the particularities of implementation frameworks or communication languages, but on the advantages and limitations of pure and emerging OO abstractions to support MAS development. The subsections below summarize the relevant features of the investigated techniques. The full description of the techniques [15, 28, 47] and methods [20, 21] is reported elsewhere since it is outside of the scope of this paper.

2.1 Pattern-Oriented Development

Pattern-oriented (PO) development [7,15] is a software engineering technique concerned with the application of design patterns. An OO pattern provides a design and implementation solution to a recurring problem, defining a set of classes and their relationships [7]. The use of design patterns has been greatly advocated in OO software development, with emphasis on reuse and maintenance. The pattern solution structures and disciplines the composition of the separated components, ensuring that the system can only change or evolve in specific, predictable ways. Patterns are the building blocks of large-scale software systems, which are likely to include instances of more than one of these patterns, composed in a variety of ways. Design patterns are not restricted to the object paradigm, but in this work we focus on OO design patterns.

In the MAS context, a pattern can be widely used as a design and implementation metaphor for recurring OO structures for MASs, thus minimizing the misalignments
between high-level agent designs and their OO detailed design and implementation. A pattern itself can be viewed as a named abstraction since it names and encapsulates guidelines for a solution in terms of a set of components, their internal elements and their interrelationships, to a recurrent problem in the MAS development. Both general patterns, such as Mediator and Composite patterns, and MAS-specific patterns [26, 44] have been used to address the traceability between agent-oriented models and OO designs.

### 2.2 Aspect-Oriented Development

Aspect-oriented (AO) software development [28, 47] has been proposed as a technique for improving separation of concerns in software construction and support improved reusability and maintainability. The aspect-oriented technique is not restricted to the object paradigm [28], but it is the focus in this experiment. The central idea is that while pure abstractions of the object paradigm are extremely useful, they are inherently unable to modularize all concerns of interest in complex systems. Thus, the goal of the AO technique is to support crosscutting concerns, by providing abstractions that make it possible to separate and compose them to produce the overall system. Crosscutting concerns are defined as system concerns that crosscut components in the design and implementation of a system.

*Aspects* are modular units of crosscutting concerns that are associated with a set of classes or objects. Central to the process of composing aspects and classes is the concept of *join points*, the elements that specify how classes and aspects are related. Join points are well-defined points in the structure and dynamic execution of a system. Examples of join points are method calls, method executions, and field sets and reads. *Advice* is a special method-like construct attached to join points. An aspect may also define attributes and methods to be introduced into the classes to which the aspect is attached. Weaver is the mechanism responsible for composing the classes and aspects. The aspect-oriented technique uses OO abstractions and aspects (components), and their internal elements: advices and join points. It also considers the 1-to-many relationship that relates one aspect to one or more classes (in this work, this relationship is named the *crosscutting* relationship). AspectJ [29] is a practical aspect-oriented extension to the Java programming language. AspectJ supports the definition of aspects, advices, join points, and pointcuts. *Pointcuts* are collections of join points and are used in advice definitions.

### 3 The Empirical Study

#### 3.1 The Methodology

The evaluation of software techniques is a notoriously hard task. There are very few established methodologies for measurement planning and data gathering. The experiment organization was based on a complementary application of the Basili’s GQM (Goal/Question/Metric) methodology [3] and our evaluation framework (Section 4) defined specially for the context of this study. GQM was used to structure the experi-
ment in terms of its goals, and the evaluation framework was defined to elicit the qualities, factors and criteria investigated in this experiment. Both were helpful in finding existing metrics and defining new ones for this empirical study.

GQM was selected to evaluate the investigated techniques as it has gained widespread popularity and support within the software engineering community. The GQM paradigm has been proposed as a goal-oriented approach for the measurement of products and processes in software engineering. This methodology is based upon the assumption that to gain a practical measure one must first understand and specify the goals of the software artifacts being measured, and the goals of the measuring process. The GQM approach provides a framework involving three steps: (1) list the major goals of the empirical study; (2) derive from each goal the questions that must be answered to determine if the goals are met; (3) decide what must be measured in order to be able to answer the questions adequately (definition of the metrics). Section 3.2 defines our goals and questions, while Section 4 presents the metrics as part of our evaluation framework.

3.2 Goals and Questions

The overall goal of this study is to evaluate the maintainability and reusability of the two investigated techniques in the MAS context. This goal was refined into a set of questions, which represent its operational definition. The idea was to generate as many questions as possible, including redundant or invalid questions. As the process proceeded, we developed a hierarchical set of questions that was subsequently narrowed down. For each question the relevant metrics were defined (Section 4). Figure 2 presents a sample of the questions generated. A report [17] presents all goals and questions that remained after refinement.

3.3 Hypothesis

The hypothesis to be tested is stated as follows: “the aspect-oriented technique provides better support for MAS maintainability and reusability than the pattern-oriented technique”. This hypothesis is based on a qualitative study, which we have conducted previously [21]. It is also directly related with the experiment goal (Section 3.2).

3.4 The MAS Project

The project upon which this system is based has been derived from a case study undertaken in the SoC+Agents/TecComm Group at PUC-Rio in Brazil (from herein referred to as Portalware). Portalware is a web-based environment that supports the development and management of Internet portals. As the needs of the Internet Portals market change even more rapidly, the frailties in the software engineering techniques that are being used become increasingly apparent. To survive, Portalware must remain extensible and modifiable, and so its design and implementation must be capable of responding to change. The agent-oriented system modeling was based on Elammari’s modeling language [10] and on the TAO modeling framework [43]. TAO was used
because it elicits common abstractions used in MAS analysis and design. UML notations [36] and the Java language were respectively used to generate the pattern-oriented designs and implementation. A UML extension for aspect-oriented design [8] and the AspectJ programming language [29] were used to generate the aspect-oriented designs and implementations.

<table>
<thead>
<tr>
<th>Goal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Evaluate the ease of evolution and reusability of the implemented multi-agent systems in order to compare the pattern-oriented development with the aspect-oriented development.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Questions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. How easy is it to evolve the system?</td>
</tr>
<tr>
<td>1.1. How easy is it to understand the system?</td>
</tr>
<tr>
<td>1.1.1. How concise is the system?</td>
</tr>
<tr>
<td>1.1.1.1. How many components (aspects/classes) are there?</td>
</tr>
<tr>
<td>1.1.1.2. How many lines of code are there?</td>
</tr>
<tr>
<td>1.1.1.3. How many attributes are there?</td>
</tr>
<tr>
<td>1.1.1.4. How many methods and advices are there?</td>
</tr>
<tr>
<td>1.1.2. How well are the agency concerns localized?</td>
</tr>
<tr>
<td>1.1.2.1. How scattered is the agenthood definition?</td>
</tr>
<tr>
<td>1.1.2.2. How scattered are the agent fundamental concerns?</td>
</tr>
<tr>
<td>1.1.2.2.1. How scattered is the autonomy concern?</td>
</tr>
<tr>
<td>1.1.2.2.2. How scattered is the interaction concern?</td>
</tr>
<tr>
<td>1.1.2.2.3. How scattered is the adaptation concern?</td>
</tr>
<tr>
<td>1.1.2.3. How scattered are the agent alternative concerns?</td>
</tr>
<tr>
<td>1.1.2.3.1. How scattered is the collaboration concern?</td>
</tr>
<tr>
<td>1.1.2.4. How scattered are the agent roles?</td>
</tr>
<tr>
<td>1.1.2.5. How scattered is the definition of agent types and their instances?</td>
</tr>
<tr>
<td>1.1.2.5.1. How scattered is the user agent type definition?</td>
</tr>
<tr>
<td>1.1.2.5.2. How scattered is the definition of a user agent’s instance?</td>
</tr>
<tr>
<td>1.1.2.5.3. How scattered is the information agent type definition?</td>
</tr>
<tr>
<td>1.1.2.5.4. How scattered is the definition of an information agent’s instance?</td>
</tr>
<tr>
<td>1.1.3. How high is the coupling of the system?</td>
</tr>
<tr>
<td>1.1.3.1. How high is the coupling between components?</td>
</tr>
<tr>
<td>1.1.4. How high is the cohesion of the system?</td>
</tr>
<tr>
<td>1.1.4.1. How high is the cohesion of the system components?</td>
</tr>
</tbody>
</table>

Fig. 2. An example of the questions generated using GQM

The MAS concerns handled by this project are the typical ones of many existing application domains of the real-world reactive MASs. This MAS encompasses several agency concerns, including agent types, roles, collaboration, interaction, adaptation, autonomy, and so on. This environment includes a number of agent types to control portals, and to coordinate and automate the time-consuming repetitive activities of the development groups. Portalware encompasses 3 agent types: (i) interface agents, (ii) middle agents, and (iii) user agents. The agent types implement the agenthood concern and other concerns. In this system, the agenthood concern is comprised of three basic agency properties: interaction, autonomy, and adaptation. Each agent instance has different agency properties and plays distinct roles. Middle agents are responsible for mediating the conversations between all system agents, providing services such as naming service.
User agents represent Portalware users and are implemented to reduce the need for cross talk between working users. Several roles are attributed to Portalware users and their respective agents, but the main ones are: (i) the Content Supplier role, and (ii) the Editor role. A content supplier (CS) is responsible for providing the portal with content segments (for instance, news). The editor is responsible for selecting from the available content segments for publishing and for assigning roles to the users. Also the agent playing an editor role has the responsibility of contacting the prospective CSs and negotiates with them the use of their capabilities. Since editors and CSs need to collaborate with each other to maintain the web portal, user agents incorporate plans for automating and supporting collaboration in different contexts. Interface agents monitor the graphical system interface in order to interact with Portalware users. They learn shortcuts by capturing preferences, and by receiving explicit instructions from the user. For instance, interface agents operate while a content supplier is authoring content segments, using keywords entered in the document and its acquired model of the user’s preferences. Interface agents do not incorporate the collaboration concern since they do not cooperate with other software agents.

The experiment subjects developed two versions of the Portalware system, using both the aspect-oriented and pattern-oriented techniques. Figures 3 and 4 represent respectively slices of the pattern-oriented and aspect-oriented designs for the Portalware MAS. Figure 3 shows a combination of different design patterns to address the MAS concerns. Each pattern is surrounded by a dotted line. In Figure 4, a diamond shape is used to express aspects. Each diamond may be related to one or more rectangles used to describe classes. This relationship is expressed as a line from the aspect to a class. Those figures also illustrate some changes required for the maintenance and reuse scenarios for further clarification in the Section 4.2.

3.5 The Subjects and Study Phases

Four subjects participated in the study. Three of the subjects were PhD candidates and one was a Master student at PUC-Rio. All of them took part in the development of the two systems. They have participated both in the development of the aspect-oriented (AO) system and in the development of the pattern-oriented (PO) system. All the subjects had experience in OO software analysis, pattern-oriented design and programming of MASs. The PhD candidates had already implemented large (> 10 thousands lines of code) Java programs. Among them, two had significant experience in aspect-oriented programming. They were asked to report problems in using the investigated techniques, and associated methods and tools. The set of agent-oriented models, detailed design and implementation were based on the same requirement specifications and satisfying the same set of scenarios.

The study was divided into two major phases: (1) the Construction phase, and (2) the Reuse and Evolution phase. In the Construction phase, the individuals were asked to develop the selected MAS (Section 3.4) using the investigated techniques. The Reuse and Evolution phase involved the same subjects. The goal of this phase was to compare the reusability and maintainability of the PO solution and the AO solution. To evaluate the modifiability and extensibility of the produced systems, a set of relevant change and reuse scenarios to both original designs and code were produced as will be described in Sections 4.2 and 5.2. Since our selected metrics (Section 4.1.2)
are oriented to the amount of design components and code lines, we had an additional standardization phase before the data collection. This phase was aimed at ensuring that the two developed MASs implemented the same functionalities. This phase also removed problems related to different coding styles.

![Fig. 3. A Subset of the Pattern-Oriented Design](image)

4 The Evaluation Framework

We developed a framework to evaluate the produced systems in terms of our defined goals, questions, and hypotheses (Section 3). The generated framework is composed of a quality model and a set of reuse and evolution scenarios. A quality model emphasizes and connects the investigated qualities, factors, and criteria [12]. It was also particularly useful for the metric selection and data interpretation. The produced systems are also evaluated through a set of scenarios generated during the case study in an attempt to verify the degree of support of the used abstractions for MAS reuse and evolution. Our evaluation framework was realized before the operation of the experiment (Section 3) and evaluated using the Kitchenham’s measurement framework [30, 31]. The detailed discussion of our evaluation framework is outside of the scope of this paper and its complete description is in [17, 42]. Section 4.1 overviews our quality model and the metrics suite, and Section 4.2 discusses briefly the reuse and evolution scenarios.

4.1 The Quality Model

Quality models are constructed in a tree-like fashion since quality is actually a composite of many other characteristics [12]. The notion of software quality is usually captured in a model that depicts other intermediary qualities, which we termed factors here. Our quality model is composed of 4 different elements: (i) qualities, (ii) factors, (iii) criteria, and (iv) metrics. The qualities are the attributes that we want primarily to
observe in the software system (reusability and maintainability). The factors are the secondary quality attributes that influence the defined primary qualities. The criteria are related to internal properties of software systems. These attributes are related to well-established software engineering principles, which in turn are essential to the achievement of the qualities and their respective factors [12]. The metrics are instruments to predict how the software artifacts achieve the selected criteria and the respective factors and qualities.

Figure 5 presents the elements of our quality model. The upper branches hold important high-level qualities and factors that we would like to quantify. The criteria are easier to measure than the factors, and thus actual metrics are connected to these criteria. The definition of our quality model was accomplished in parallel with the use of GQM methodology (Section 3). The following subsections describe the elements of our quality model.

4.1.1 The Qualities and Factors
The first obligation of any software-measurement activity is the identification of the primary quality attributes and software artifacts we wish to measure. This study focuses on the evaluation of the maintainability and reusability attributes based on distinct artifacts of a MAS, such as its design models and source code.

**Reusability and Maintainability.** Reusability is the ability of software elements to be used for construction of other different elements in the same software system or across different ones [34]. In this study, we are interested in evaluating the reusability of agency concerns in design and code. Maintenance is the activity of modifying a software system after initial delivery. Software maintainability is the ease with which the software components can be modified. Maintenance activities are classified into four categories [12,45]: corrective maintenance, perfective maintenance, adaptive maintenance, and evolution. This work focuses on MAS evolution, i.e. the addition or elimination of agency concerns.
**Factors.** The model emphasizes that similar factors are useful for the promotion of maintainability as well as reusability. This similarity is related to the fact that the reuse and maintenance activities encompass common cognitive tasks. Flexibility, understandability, modifiability, extensibility are the central factors to promote reuse and maintainability [12,34, 40,45]. Both kinds of activities require software abstractions to support understandability and flexibility. Understandability indicates how easy it is to study and understand the design and the code of a system [40]. Flexibility indicates how easy it is to make drastic changes to one component of a system without a need to change others [40]. An understandable system enhances its own maintainability and reusability because most maintenance and reuse activities involve software engineers when trying to understand the system components prior to any further system modification or extension. In addition, a software system needs to be flexible enough to support the addition and removal of functionalities and the reuse of its components without making a lot of effort. The modifiability and extensibility factors also influence maintainability and reusability and are measured based on the scenarios presented in Section 4.2.

### 4.1.2 The Metrics

The metrics for Size, Coupling, Cohesion and Separation of Concerns were selected and defined to evaluate the techniques and produced systems with respect to the degree of system maintainability and reusability supported. We have reused and refined some classical and OO metrics [9,11,12] and have defined new ones that capture important notions for the purpose of this study. Although a large body of research in software metrics has focused on procedural or OO software, there is no software metric for aspect-oriented software until now. So we tailored some object-oriented metrics to apply them to aspect-oriented software. In fact, we reworked the definition of the metrics since we should be able to compare OO designs and code with aspect-oriented designs and code. Each metric definition was extended to be applied in a way
that is independent from the paradigm, supporting the generation of comparable results.

Furthermore, we have proposed some metrics for separation of concerns, which are described elsewhere [17,42]. The separation of concerns metrics measure the degree to which a single concern in the system maps to the design components, operations, and lines of code. The more directly an agency concern maps to such code and design units, the easier it is to understand it. The more directly a concern maps to the units, the fewer units will be modified during maintenance activities or the fewer units need to be understood and extended during reuse activities. The definition of our separation of concerns metrics was inspired in Lopes’ work [33]. Table 2 associates the metrics with the QGM questions and the criteria defined by the quality model. Table 2 also presents the sources for the metrics which we have based on.

<table>
<thead>
<tr>
<th>Metrics</th>
<th>Answered Questions</th>
<th>Criteria</th>
<th>Based on</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vocabulary Size (VS)</td>
<td>How many components are there?</td>
<td>Size</td>
<td>Papaioannou [39]</td>
</tr>
<tr>
<td>Lines of Code (LOC)</td>
<td>How many lines of code are there?</td>
<td>Size</td>
<td>Fenton [12]</td>
</tr>
<tr>
<td>Number of Attributes (NOA)</td>
<td>How many attributes are there?</td>
<td>Size</td>
<td>Chidamber [9]</td>
</tr>
<tr>
<td>Weighted Operations per Component (WOC)</td>
<td>How many methods and advices are there?</td>
<td>Size</td>
<td>Chidamber [9]</td>
</tr>
<tr>
<td>Coupling Between Components (CBC)</td>
<td>How high is the coupling between components?</td>
<td>Coupling</td>
<td>Chidamber [9]</td>
</tr>
<tr>
<td>Lack of Cohesion in Operations (LCOO)</td>
<td>How high is the cohesion of the components?</td>
<td>Cohesion</td>
<td>Chidamber [9]</td>
</tr>
<tr>
<td>Depth of Inheritance Tree (DIT)</td>
<td>How high is the coupling between components?</td>
<td>Coupling</td>
<td>Chidamber [9]</td>
</tr>
<tr>
<td>Concern Diffusion over Components (CDC)</td>
<td>How well are the agency concerns localized?</td>
<td>Separation of Concerns</td>
<td>Lopes [33]</td>
</tr>
<tr>
<td>Concern Diffusion over Operations (CDO)</td>
<td>How well are the agency concerns localized?</td>
<td>Separation of Concerns</td>
<td>Lopes [33]</td>
</tr>
<tr>
<td>Concern Diffusions over LOC (CDLOC)</td>
<td>How well are the agency concerns localized?</td>
<td>Separation of Concerns</td>
<td>Lopes [33]</td>
</tr>
</tbody>
</table>

4.2 The Reuse and Evolution Scenarios

We have simulated simple and complex changes involving agency concerns to both the OO and AO solutions in order to measure their modifiability and extensibility support. We have selected 7 change and reuse scenarios which are recurrent in large-scale MASs, such as inclusion of new agents, reuse of roles and collaborative capabilities, and so forth. The list of the scenarios is as follows:

S1) Change on the Agent Roles (Evolution)
S2) Creation of an Agent Type (Evolution)
S3) Reuse of the Agenthood Concern (Reuse)
S4) Inclusion of Collaboration in an Agent Type (Reuse and Evolution)
S5) Reuse of Roles (Reuse)
S6) Creation of a new Agent Instance (Evolution)
S7) Change of the Agenthood Definition (Reuse and Evolution)

For each change made to the system concerns, the difficulty of the concern modifiability is defined in terms of the following items: (1) number of components (aspects/classes) added, (2) number of components changed, (3) number of relationships included, (4) number of relationships changed, (5) number of new LOCs, (6) number of modified LOCs, (7) number of operations (methods/advices) added, and (8) number of operations (methods/advices) changed.
For each attempt made to reuse some concern, the difficulty of extensibility is measured based on the modifiability items and on the following additional items: (9) number of copied entities, and (10) number of copied LOCs. All these items were observed from the structural and behavioral design models and code points of view. Those elements in aspect-oriented development and those in pattern-oriented development are comparable because they represent the same concerns of high-level agent models.

5 Results

This section presents the results of the measurement process. Section 5.1 presents an overview of the results obtained at the end of the MAS construction phase. The data have been collected based on the set of defined metrics (Section 4.1.2). The complete description of the data gathered in the construction phase is reported in [17]. Section 5.2 describes the results of the MAS evolution and reuse phase based on the selected scenarios and associated metrics (Section 4.2). The discussion about the results is concentrated in Section 6.

5.1 The MAS Construction Phase

The data was partially gathered by the CASE tool Together 6.0. It supports some metrics: LOC, NOA, WOC (WMPC2 in Together), CBC (CBO in Together), LCOO (LOCOM1 in Together) and DIT (DOIH in Together). Figure 6 presents the overall results of the AO and PO projects for all of the metrics, excluding the separation of concerns metrics. These results were gathered from the system viewpoint. The number of system components in the PO project was 7% higher than in the AO project (VS metric). The tallies of lines of code (LOC) and number of attributes (NOA) for the developed MAS in the PO implementation were respectively 12% and 9% higher than in the AO code. The AO project also produced better results in terms of complexity of operations (6%), component couplings (9%), and component cohesion (3%). The maximum DIT was five for the PO project and three for the AO project, that is, 40% lower than in the PO project.

The absolute measurements gathered from the system perspective (Figure 6), show significant differences in favour of the aspect-based solution. However, it was not always the same when analyzing the results under different viewpoints. The following subsections present in detail the results of the size metrics (Section 5.1.1), the coupling and cohesion metrics (Section 5.1.2), and the separation of concerns metrics (Section 5.1.3).

5.1.1 Results of the Size Metrics

Vocabulary Size. VS counts the external vocabulary of the system, i.e. the number of components (classes/aspects) in the system. The external vocabulary of the aspect-oriented MAS is simpler than in the pattern-oriented MAS, since the amount of design and implementation components in the latter (VS = 60) was higher than in the
The main reason for this result is that the Role and Mediator patterns required additional classes to address the decomposition and composition of multiple agent roles and behavior properties, respectively. The aspect-oriented solution does not need such additional classes since the composition is specified by the pointcuts, which are defined internally to the aspects.

**Number of Attributes.** This metric counts the internal vocabulary of each component, i.e. the number of attributes of each class or aspect. Inherited attributes are not included in the tally. The internal vocabulary of the PO solution components is more complex than in the AO components. The number of attributes of the MAS components in the PO solution was higher than in the AO solution. The reason is because the Agent objects in the PO project need to have explicit references to the objects representing the three agent basic properties and the agent roles. For example, the Agent class (the class that encapsulates the agenthood structure and behavior) is composed of 9 attributes in the PO system, while it has 6 attributes in the AO system. In addition, the collaboration concern is modularized directly by the Collaboration and Role aspects, while it is spread over 5 different classes (Collaboration, CollaborationCore, CollaborationRole, CollaborativeAgent, and Role) in the PO project and, as a consequence, this is an increasing factor in the NOA since each of these classes need to have references to each other.

**Lines of Code.** The LOC was 1445 in the PO implementation and 1271 in the AO implementation, that is the PO code has 174 lines more than the AO code. In general, the implementation of each (plan, role) pair in the PO code included 10 lines more than the AO code. The reason is that the PO solution needs additional method calls on

![Fig. 6. Comparison of the Results of the Two Projects](image-url)
the Role subclasses to: (i) activate and deactivate roles, and (ii) to get references to the role objects. These calls are not needed in the aspect-oriented project because the composition is specified by the pointcuts in the role aspects. In addition, fewer code lines were needed to implement the collaboration concern in the AO system since the PO system implements additional classes of the Role pattern, as described previously.

**Weighted Operations per Component.** This metric measures the complexity of a component in terms of its operations. The higher the number and complexity of the component operations, the more difficult it is to understand the system. In general, these measures highlighted higher WOCs in the AO solution. It happens because the modularization of some concerns in the aspects requires the context to be recaptured by means of join points and pointcuts. Some examples of the WOC measures are: (i) 9 for the Adaptation class in the PO solution against 10 for the Adaptation aspect in the AO solution, (ii) 12 for the Autonomy class against 14 for the Autonomy aspect, and (iii) 17 for the Interaction class against 25 for the Interaction aspect. The recurrent situation in these examples is that the Interaction, Adaptation and Autonomy aspects receive the agent object reference as parameters in the pointcuts, while such reference is accessed in a local attribute in the respective classes of the PO project.

### 5.1.2 Results of the Coupling and Cohesion Metrics

**Depth of the Inheritance Tree.** DIT is defined as the maximum length from the node to the root of the tree. There are major problems in the PO design when considering this metric. The use of the role pattern leads to a 5-level hierarchy to structure the agent roles, which potentially leads to a high inheritance coupling [45]. In the AO solution, the DIT = 1 since roles are encapsulated in aspects which crosscuts the agent type hierarchy. Moreover the use of the mediator pattern leads to a 3-level hierarchy to structure the agent types, while the same hierarchy in the AO system has 2 levels. The reason is that it is necessary to create an additional level (the CollaborativeAgent class) in the PO solution in order to separate the collaborative agents from the non-collaborative ones. In the AO solution, this separation is accomplished transparently by the collaboration aspect that defines which agent types are collaborative and, as a consequence, it does not affect the size of the agent type hierarchy.

**Coupling between Components.** This metric is defined for a class or aspect as a tally of the number of other classes and aspects to which it is coupled. There is a significant difference between the CBC of the two solutions for the Agent class. The CBC is 12 in the PO system and 9 in the AO solution. This time, the reason for the difference is that the Agent class needs to have explicit references to the classes representing the basic agent properties. The reverse is also necessary, that is the Autonomy, Interaction, and Adaptation classes require explicit references to the agent object. In the AO project, only the aspects have reference to the Agent class.

**Lack of Cohesion in Operations.** This metric measures the cohesion of a component. A low LCOO value indicates high coupling between methods (i.e. high cohesion) which is a desirable situation. Low-cohesive components suggest an inappropriate design, since it determines the encapsulation of unrelated program entities and that should not be together [2]. This also indicates potentially less reusable and maintainable components. Most components of the PO system produced better results in terms
of cohesion than the components of the AO system. For example, the LCOO for the Agent class is 50 in the PO project and 57 in the AO project. The LCOO for the Interaction concern is 29 in the PO project and the 48 in the AO project.

5.1.3 Results of the Separation of Concerns Metrics

**Concern Diffusion over Components.** Every MAS concern required more components in the definition of the PO solution than in the AO solution. All roles required more than 5 classes to their definition, while one single aspect is able to encapsulate each system role: Answerer (6 against 1), Caller (7 against 1), ContentSupplier (6 against 1), and Editor (7 against 1). The agency properties also needed more components in the PO design and implementation: adaptation (3 against 1), autonomy (3 against 2), collaboration (15 against 6), and interaction (7 against 6). Finally, more components are also used in the PO design and implementation of agent types – 37 against 33 both for information and user agent types – and their respective instances – 3 against 1 for information agents and 4 against 2 for user agents.

**Concern Diffusion over Operations.** Again all concerns require more operations (methods/advices) in the PO system than in the AO system. Most concerns in the PO solution were implemented with more than the double of the number of operations used in the AO system. For instance the case for the Adaptation concern and the Autonomy concern. There are some cases where the difference is even bigger (for example, all roles), less than the double (for example, collaboration and both agent types), or almost the same (for example, agenthood and Interaction).

**Concern Diffusion over LOC.** The measures here also pointed out to the fact that the AO solution was more effective in modularizing the MAS concerns. The results for the agent types did not show any difference. All other cases, except the agenthood concern, were better encapsulated in the AO design and implementation. The detected differences were very significant in the following concerns: (i) the basic agent properties, (ii) the agent roles, (iii) the agent instances, and (iv) the collaboration property. The problem with the agenthood concern occurred since the Interaction aspect specifies in its definition that it must to be executed before the Collaboration aspects, increasing the number of transition points.

5.2 The MAS Evolution and Reuse Phase

In the following, we discuss each scenario (Section 4.2) and respective results in both pattern-oriented and aspect-oriented solutions. The gathered data here confirmed most the problems identified in Section 5.1. Table 3 overviews the main results of the evolution and reuse scenarios. Figures 3 and 4 illustrate some basic changes in the design artifacts, which were necessary in the scenarios S2 and S4.

**Scenario S1** – Change on the Agent Roles

A new role and respective plans were incorporated to the system in order to encapsulate a reviewer behavior and associated plans. This role must be associated with the UserAgent type that is the agent type that models the system users. This change resulted in similar impact to both PO and AO solutions. However, some additional lines were necessary in the change of the PO system.
Table 3. The Results of the Reuse and Maintenance Scenarios

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Changed Comp.</th>
<th>Changed Operat.</th>
<th>Added Comp.</th>
<th>Added Operat.</th>
<th>Changed Relations</th>
<th>Added Relations</th>
<th>Added LOCs</th>
<th>Changed LOCs</th>
<th>Copied Comp.</th>
<th>Copied LOCs</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>5</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>S2</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>S3</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>S4</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>8</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>S5</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>S6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>S7</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

Scenario S2 – Creation of an Agent Type

Portalware users often need to search for information that is stored in multiple databases and available on the web in order to produce the material required by the editors. As a consequence, a new agent type, called InformationAgent, was included into the system for automating this time-consuming task. Each InformationAgent instance contains different searching plans and is attached to an information source. This change resulted in similar impact to both PO and AO solutions. However, this scenario triggered two additional change scenarios (S3 and S4) which are described immediately below.

Scenario S3 – Reuse of the Agenthood Concern

Since all system agent types incorporate the agenthood features, the incorporation of the InformationAgent type required the reuse of these features. As presented in Table 3, this scenario also resulted in similar changes to both PO and AO systems.

Scenario S4 – Inclusion of Collaboration in an Agent Type

The inclusion of the information agent also required the reuse of the collaboration concern previously defined in the system. It is because information agents collaborate with each other when an information agent is not able to find the required information in the respective information source. As a result, information agents play the caller and answerer roles respectively to: (1) call another information agents and ask for an information, and (2) receive requests and send the search result to the caller. Therefore this scenario included two main tasks: (1) the reuse of the predefined collaboration concern in the context of the InformationAgent, and (2) creation and attachment of two roles to the InformationAgent type. This was the scenario that resulted in more substantial differences between the changes in the PO solution and the AO solution: (1) the PO code required 20 lines more than the AO code, (2) more relationships were added in the PO design, and (3) eight lines were removed from the AO code while no line was changed in the PO code.
Scenario S5 – Reuse of Roles
The introduction of information agents to the system made information services available to the other agents. Since some user agents play the content supplier role, the system could automate the task of selecting information relevant to certain contexts on behalf of their respective users. In this sense, the user agents should use the services of the information agents playing a caller role, which is already defined in the MAS. In other words, the caller role is supposed to be reused and attached to UserAgent type and the ContentSupplier role in both AO and PO solutions. This reuse scenario required more effort in the PO project than in AO project in the following ways: (1) number of changed operations (2 against 1), (2) number of added relationships (4 against 2), and (3) number of added LOCs (16 against 14).

Scenario S6 – Creation of a new Agent Instance
This scenario investigated the impact of adding new agent instances into the system. In particular, we have created a new instance of the UserAgent type to play the content supplier role. This maintenance scenario required the same changes in the PO and AO versions of the MAS. The addition of 15 LOCs was implemented in the two versions.

Scenario S7 – Change of the Agenthood Definition
The last scenario was created to simulate a really pervasive change in both solutions. With the inclusion of the information agent into the system, all other agent types are able to use its services to achieve their specific goals. The agenthood definition must be extended to include the collaboration concern. So every agent type should be collaborative which implies in removing the collaboration concern from the UserAgent class and associate the collaboration component with the Agent class. So, this scenario requires the reuse of the previous collaboration definition in a new context. The AO solution provided better modifiability and extensibility support in this case: (1) 5 components were changed in the PO design and just one in the AO design, (2) 5 relationships were changed in the PO design while 2 were modified in the AO design, (3) 40 lines were changed in the PO code and no line was modified in the AO code.

6 Discussion and Related Work
Although the conclusions cannot be extrapolated to all MASs, this study was conducted in a system that includes the canonical features of reactive MASs. A number of procedures were considered in order to minimize common problems in software engineering experiments. In addition, the present work provides an experimental framework (and selects a set of MAS concerns) which other MASs developers can reuse and refine in their MAS applications in order to improve the knowledge of the software engineering community about the interplay between agents and objects, and the associated difficulties in designing and implementing MAS with OO techniques.

We cannot guarantee that the used aspect-oriented and object-oriented solutions are the best ones for the problem at hand. However, the selected methods base their design guidelines on well-documented solutions [15, 20, 21, 26]. In the following, we provide a detailed description of the results and an analysis for the stated hypothesis.
We also discuss how our comparative study provides software engineers with a better understanding of the interplay between agents and objects. Finally, we describe the constraints on the validity of this evaluation.

6.1 Techniques Comparison

In general, the results (Section 5) have confirmed our hypothesis. The AO technique provided better maintenance and reuse support for the selected project (Section 3.4) than the PO technique. Even thought the aspect-oriented project was carried out before the pattern-oriented project, most measures indicated that the former solution presented better results than the latter one. The aspect-oriented technique produced a more concise MAS, in terms of lines of code, external vocabulary and internal vocabulary of the components (Section 5.1.1). The use of design patterns leads to an increase in the number of classes, which became dedicated to overcome the limitations of the composition mechanisms of OO programming languages. This conclusion is supported by all size metrics, except the WOC metric.

The aspect-oriented technique produced more complex operations, i.e advices, than the object-oriented technique (Section 5.1.1). It occurs because the modularization of some concerns in the aspects requires the object contexts to be recaptured by means of pointcuts. This result is supported by the WOC values. The pattern-oriented technique also leads to the abuse of the inheritance mechanism, which is fundamental for the establishment of high inheritance couplings. This problem was detected by the DIT values (Section 5.1.2).

The pattern-oriented technique produced more highly coupled components than the aspect-oriented technique. This is a consequence of the lack of expressive power of this technique to modularize MAS concerns. This result is supported by the CBC metric (Section 5.1.2). Moreover the pattern-oriented technique produced better results in terms of cohesion than the aspect-oriented technique (Section 5.1.2). The lack of cohesion in the aspects occurs because an aspect is meant to encapsulate crosscutting behavior applied to different components. However, these behaviors cannot be directly related to each other, producing high LCOO values.

The aspect-oriented technique clearly provides better support for separation of MAS concerns. The aspect-oriented mechanisms provided improved support for the modularization of agency concerns. This finding is supported by all separation of concerns metrics (Section 5.1.3). Finally, in terms of the evolution and reuse scenarios (Section 5.2), the aspect-oriented technique presented better results. This conclusion is supported by the following metrics: changed components, changed relationships, added relationships, added LOCs, and copied LOCs.

6.2 Agents vs. Objects

In general, we found that advanced OO techniques, supported by effective methods, can deal successfully with concerns of reactive MASs. The additional abstractions (patterns and aspects) were important to cover conceptual gaps between agents and objects, as discussed as follows. The pure OO abstractions were useful to encapsulate some basic agency concerns. Classes and the inheritance mechanism provide direct
support for structuring the multiple agent types. Inheritance was interesting to promote reuse of operations common to all agent types, such as belief and plan updates.

Patterns and aspects minimized the misalignments between software artifacts. Both techniques were very helpful to promote traceability among agent-oriented models and OO designs and code. However, the aspect notion provided better traceability since when an agent property was added or removed to the agent-oriented model, this was directly accomplished in the aspect-oriented design and code segments (Section 5.2). The aspect abstraction is also more appropriate to deal with roles from the reuse and maintainance viewpoints (Section 5.2). These findings are similar to the ones reported in [25].

Design patterns have no first-class representation at the implementation level. The implementation of a design pattern, therefore, cannot be reused and, although its design is reused, the MAS developer is forced to implement the pattern many times. Unlike patterns, aspects provide first-class representation at the implementation-level for agency concerns (such as interaction, collaboration, roles, and so on), supporting reuse both at the design and implementation levels. For example, the reuse of the Collaboration property in the context of user agents required the association of the Collaboration aspect to UserAgent class, depicting the join points of interest, while in the pattern-based approach, some additional modifications (Table 3) were required to introduce the association as well as the explicit calls to methods defined in the interface of the Collaboration class.

Although the results have shown that the aspect-oriented approach provided better reusability and maintainability, the pattern-oriented approach also supported well some MAS concerns. So the aspects and OO patterns can naturally co-exist in a hybrid solution. In [41] Rashid presents an example of a hybrid approach to separation of concerns.

6.3 Threats to Validity and Related Work

Our study encompasses some validity constraints in terms of construct validity, internal validity and external validity. According the construct validity, maintainability, reusability, understandability and flexibility are difficult concepts to measure. The dependent variables used here (Section 4.2) are based on a previous study performed by Li et al. [32]. In their work, the concept “maintenance effort” was reified as the number of lines of code changed. In future work, we are planning to use other more representative measures, such as “time to understand, develop and implement modifications” [1]. The achievement of the construct validity for our independent variables (the metrics in Section 4.1.2) would be desirable but is beyond the scope of this paper. However, some of our metrics are extensions of CK metrics that have been theoretically validated.

With respect to internal validity, our empirical study cannot be considered a controlled experiment, since all subjects took part in the development of the two systems. However, we tried to minimize the bias, selecting two subjects who defended (before the study) the pattern-oriented approach and two others who defended the aspect-oriented approach. Despite the effect caused by the subjects learning as the study proceeded, the AO system, which was developed first, showed better results.

Finally, with respect to external validity, the limited size and complexity of the system and the use of student subjects may restrict the extrapolation of our results.
However, while the results may not be directly generalized to professional developers and real-world systems, the academic setting allows us to make useful initial assessments of whether the investigated techniques would be worth studying further. In spite of its limitations, the study constitutes an important initial empirical work on the investigated techniques and is complementary to a qualitative work [21] that we performed previously.

Up to now, most empirical studies involving aspects rest on subjective criteria and qualitative investigation [21,22,27]. For example, Hannemann and Kiczales compare Java implementations and AspectJ implementations of the GoF design patterns [22] in terms of weakly defined measurement criteria, such as composability and pluggability. Only a few papers use software metrics for aspect-oriented software development, such as Lopes’ work [33]. She has defined a set of different metrics for separation of concerns. In fact, our metrics CDC, CDO and CDLOC are somewhat inspired by her set of metrics. However, the Lopes’ metrics only capture different dimensions of separation of concerns. In addition, the definition of her study is quite strongly coupled to the distribution concern in Java code. Our suite of metrics [42] generalizes her metrics to apply to different concerns of design and code. Moreover, our metrics are early prediction mechanisms for other stringent principles in the design of aspect-oriented software, such as coupling and cohesion.

7 Conclusions and Ongoing Work

The separation of MAS concerns is essential to MAS engineers since they may decide to extend and modify such concerns as the system evolves [18]. The development of complex MASs undergoes a transition from agent-oriented models into object-oriented design and implementation. This transition needs to ensure that the MAS concerns, which are encapsulated by abstractions in high-level agent-oriented models, are also successfully mapped into abstractions available in OO programming languages and implementation frameworks. Among the problems inherent to such transition, none is more serious than the difficulty to handle the conceptual differences between agents and objects, requiring the application of well-established principles and their supporting techniques and methods. More generally speaking, there is a need for understanding the relationships between objects and agents.

In a previous work [43], we have identified several commonalities and differences between agent-oriented and object-oriented abstractions within a unified conceptual framework. However, we have not made it explicit how concerns to MAS engineers are successfully modularized with OO abstractions in the design and implementation phases. This paper has presented an empirical study designed to understand this phenomenon and compare the maintenance and reuse support of two emerging OO techniques for MAS development. The results have shown that the aspect-oriented technique allowed the construction of a reactive MAS with improved structuring for reuse and maintenance of the MAS concerns. The use of aspects resulted in better separation of concerns, lower coupling between its components (although less cohesive), and fewer lines of code. Another important conclusion of this empirical study is that the aspect-oriented approach also supported a better alignment with the higher-level abstractions from agent-oriented models. Since this is a first exploratory study, to further confirm the findings, other rigorous and controlled experiments are needed.
We are currently developing a similar empirical study for both an open electronic marketplace system and an urban traffic management system which involve a substantial number of types of mobile and cognitive agents.

Acknowledgements

We would like to thank Gail Murphy for her valuable contributions during this work. This work has been partially supported by CNPq under grant No. 141457/2000-7 for Alessandro, grant No. 140646/2000-0 for Viviane, and by FAPERJ under grant No. E-26/150.699/2002 for Alessandro. Cláudio is supported by CAPES under grant No. 3100512004M-9. The authors are also supported by the PRONEX Project under grant 7697102900, and by ESSMA under grant 552068/2002-0 and by the art. 1st of Decree number 3.800, of 04.20.2001.

References


Architecting the Design of Multi-agent Organizations with Proto-frameworks

Andrés Díaz Pace*, Marcelo Campo*, and Alvaro Soria

ISISTAN Research Institute, Facultad de Ciencias Exactas, UNICEN University
Campus Universitario, Paraje Arroyo Seco, B7001BBO Tandil – Argentina
{adiaz,mcampo,asoria}@exa.unicen.edu.ar

Abstract. The popularity of multi-agent systems (MAS) is on the rise. However, the provision of guidelines to assist the developer to manage both agency and organizational concerns, so that these concerns can be mapped to framework implementations, is still a technically difficult task. In this context, this work describes a design approach based on the notion of proto-frameworks, which proposes the materialization of MAS architectural models into object-oriented structures, enabling then different implementation alternatives for MAS frameworks according to specific developer’s needs.

1 Introduction

Over the last years, many research proposals have increasingly embraced multi-agent systems (MAS) as a viable approach for engineering a wide range of systems [14]. The popularity of MAS has led to an emerging research community [13], which is focused on the study of software engineering techniques for MAS development.

Basically, one can think about agents as architectural components with two central properties: openness and autonomy, plus an underlying organizational structure. Agreeing on that agents represent useful abstractions for the design of large-scale, distributed software is a good starting point, however, like any piece of software, agents must be managed in order to satisfy a number of quality attributes such as scalability, maintainability, dependability and reusability, among others. From a design perspective, we can find two principal reasons for MAS complexity:

- Developers have usually to understand, choose and combine a broad set of agency features (e.g., perception, action, learning, communication, mobility, etc.) in order to successfully develop individual agents [12].
- When considering a MAS as a group of interrelated agents that exhibit some kind of social behavior, developers have also to select appropriate organizational structures [28]. As argued in [9], the application of predetermined (architectural?) organizational metaphors in the design of MAS comes often associated with certain parameters, which will ultimately influence the predominant quality attributes of the resulting system.

* Also CONICET- Argentina
To tackle the aforementioned problems, many approaches have proposed the use of object-oriented frameworks [11] to express both agency features and organizational concerns [5, 16, 18, 22, 29]. This is based on the argument that frameworks can be suitable instruments to describe reusable designs and capture the essence of (agent-oriented) patterns, components and architectures. In recent times, some authors have informed about the partial adequacy of conventional framework approaches to deal with the abstractions required by MAS [27], though, just sparse solutions have been reported [14]. Besides, these approaches have paid little attention to organizational issues. Therefore, the provision of guidelines to help developers bridge the gap between MAS models and frameworks constitutes a very important, but still rather unexplored, aspect of MAS development techniques.

In this context, the authors have developed in prior research [6], a design method based on the notion of object-oriented materialization of software architectures derived from architectural styles [1, 23]. The approach introduces a novel concept called proto-framework, which is an “architectural” framework that provides the basic bricks to build new frameworks that adopt its underlying architecture. A remarkable aspect of this research is that the approach is general enough to be applied in MAS design as well [7]. In this work, we argue that proto-frameworks can serve specifically to the design of multi-agent organizations, by allowing the mapping of agent models expressed in terms of architectural models to suitable MAS frameworks. Furthermore, this mapping can promote different implementation alternatives of these frameworks according to specific developer’s needs.

The rest of the paper is divided into 5 sections. Section 2 discusses the issue of architectures and frameworks for MAS organizations. Section 3 briefly introduces our architecture-driven framework approach and presents its contribution to organizational abstractions, using the concept of proto-frameworks. Then, section 4 describes an example of an architectural model called Bubble and its proto-framework counterpart. In this context, section 5 shows the proto-framework in action, with a case-study based on development of a real-life MAS framework called InQuality. Section 6 analyzes related lines of research. Finally, section 7 rounds up the conclusions of the paper.

2 Developing MAS as Computational Organizations

The construction of MAS, from their conception to specific implementations, involves generally a number of analysis and design activities [3, 29]. In order to facilitate this process, different models have been proposed by the agent community [3, 27]. Very briefly, we can say that these models outline a general structure for the system being developed, as well as the relationships among the constitutive components. In organizational terms, these models should define the roles of the various components, their responsibilities for tasks and goals, the allocation of resources, and a set of communication protocols for inter-agent interactions to follow [28]. As argued by Zambonelli et.al. [28], an organizational metaphor can make the design of
Architecting the Design of Multi-agent Organizations with Proto-frameworks

MAS less complex and easier to manage than more traditional approaches for distributed systems. This claim is attributed to a cleaner separation of concerns between the intra-agent and the inter-agent dimensions [12]. A graphical characterization of MAS under this organizational conception is depicted in Figure 1.

Fig. 1. Organizational characterization of MAS (extracted from [28])

Nonetheless, the design of MAS organizations is not straightforward. Generally, existent approaches to organizations for MAS are strongly based on role models, but the organizational structure is only implicitly defined by the models. This precludes developers of fully exploiting patterns for these structures at the design phase. To overcome this limitation, Zambonelli et.al. [28] have introduced three novel organizational abstractions, namely: organizational rules, organizational structures and organizational patterns. In addition, the authors have sketched the main inputs/outputs of a MAS methodology based on these abstractions. The work mostly deals with analysis aspects and their relationship with organizational rules, but gives few guidelines with respect to design support for the proposed abstractions, and especially for organizational structures. The goal of this paper is to take a step forward in this direction, addressing this aspect by means of a design approach driven by architectural mechanisms and frameworks.

2.1 Towards an Architectural Vision of Multi-agent Design

The software architecture paradigm [1, 23] focuses on the solution of design problems taking as main driver the organization of software components in function of the quality attributes affecting the system. The paradigm builds on a number of architectural patterns [4, 23], which capture typical ways of structuring software systems to fulfill specific quality attributes. There are also variants of these patterns that address
specific MAS concerns [9, 26]. Designers usually construct their systems by assembling and elaborating these predefined architectural fragments.

With a software architecture as a cornerstone, it is desirable to have mechanisms to map the components prescribed by the architecture into code structures. Architectural requirements may admit multiple alternative realizations, with different characteristics regarding quality attributes such as performance, modifiability or security, among others. From an object-oriented perspective, frameworks [11] have good credits to refine architectural elements. In the case of MAS, a framework can permit developers to extend/adapt certain components in order to construct various types of agents with different capabilities and patterns of interaction. Along this line, there is a diversity of MAS environments designed according to these ideas (e.g., Madkit, Jafmas, Jafima, BrainstormJ, and Zeus, to enumerate a few) [5, 16, 18, 22, 29].

Analyzing MAS frameworks, it has been observed that whereas they tend to be well-suited for the construction of units encapsulating features related to inter-agent issues (e.g., perception, action, representation and manipulation of knowledge, communication, etc.), they often fall short when dealing with organizational issues. Even though organizational structures have a representation at the framework level, we advocate that their definition is mostly done in terms of architectural elements, and this poses organizational models at the architectural level. At this point, once the system architecture is given, the underlying organization of its concerns comes attached. Hence, the adoption of the right organizational structures at the architectural level will prescribe the kind of organizational support that later frameworks should accommodate to. The main consequence of this line of reasoning is that an architectural model can be seen as a special organizational model, structured in terms of components that require services from each other via connectors, plus a set of constraints and valid configurations.

3 Object-Oriented Materialization of Architectures Using Proto-frameworks

Architecture-based design can be seen as an activity that takes an initial architectural description as input, and progressively decomposes and refines this description until a more detailed design specification is obtained. The architecture will provide the fundamental abstractions for the domain, so that the relevant quality attributes are satisfied, and also a set of composition mechanisms to assemble the resulting abstractions. In this context, an architecture does not always prescribe an object-oriented computational solution (object orientation is just a particular architectural style that indicates the usage of objects as architectural components and messages as connectors [23]). Anyway, objects can be a very convenient technology to implement, or materialize, different architectural styles. The key idea of the proposed approach relies on the notion of object-oriented materialization of domain-specific architectures.

By materialization, it is meant the process of producing a concrete computational representation from an abstract architectural description (e.g., expressed in an ADL
using a given technology (in this case, objects). More specifically, the approach transforms components and connectors at the architectural level into classes, methods and class relationships at the object-design level. In addition, given the generic characteristics of architectural models, it results quite natural to think about a framework implementation support for the object materialization of these models. This kind of “architectural” frameworks, called proto-frameworks, are expected to work nearly as a bridge between the architectural and the object contexts.

In a proto-framework [6], the essential abstractions of the domain are captured by the framework’s classes, providing the entire infrastructure needed for cooperation and communication of each component type. This gives very abstract hooks to map specific domain components into a class hierarchy in a white-box fashion. Then, the instantiation process can produce a specific application, but yet more important, it can lead to new domain-specific frameworks that adopt the underlying architectural model. In both cases, the tradeoffs among quality attributes selected by the designer for a target proto-framework can generate different implementation alternatives. Therefore, the conceptual architecture becomes more explicit and visible both in terms of design and code artifacts

3.1 Outline of the Materialization Process

In general, the development of architecture-driven frameworks based on proto-frameworks can be seen as a recursive process of decomposition involving both architectural structures and coarse-grained functionality, as suggested by the ADD method [2]. It proceeds as follows:

1. **Analysis of architectural alternatives for the domain.** The process starts initially with the definition of an abstract domain model and the identification of one or more architectural styles [4, 23] (e.g. pipes and filters, blackboard, hierarchical layers, etc.) adequate for the domain.
2. **Architectural description.** From the precedent analysis, a first architectural representation with the main components and their relationships is obtained. This architecture can be specified, for example, with any available ADL (see [20] for a survey).
3. **Derivation of a first architectural materialization (proto-framework).** A basic proto-framework is generated, by translating the architectural description to object-oriented constructs. This materialization is driven, essentially, by the quality attributes that predominate in the domain.
4. **Application of design patterns.** The process continues with the refinement of the proto-framework. In such an activity, design patterns or other design mechanisms can be applied to obtain a more flexible design.
5. **Development of more elaborated frameworks or applications.** Here, specific methods/subclasses must be defined, in order to capture the control flow and the skeleton of the application respectively. Some points that can not be generalized can be traduced to abstract and hook methods.
3.2 Adding Support for Organizational Structures

As mentioned before, a software architecture gives a coarser level of granularity for grasping the design of a system (no matter if it is MAS-oriented or not), independently of implementation technologies. To support the position that organizational abstractions can be reasonably allocated to architectural models, we have extended the approach in order to support organizational concerns (see Figure 2).

![Fig. 2. Extending object-oriented materialization with organizational abstractions](image)

The modifications to the process given in the precedent sub-section are summarized below.

- **Domain analysis is enhanced with organizational rules and structures.** The incorporation of rules and structures in analysis provides a preliminary organizational blueprint to the architecture, though there is no commitment yet to definitive organizational structures for the architectural design.

- **MAS patterns are treated as specialized instances of traditional architectural styles.** The choice of organizational structures entails ultimately the selection of predetermined mechanisms codified as architectural styles. This uniform view leads to more open MAS architectures, not necessarily built on the basis of MAS technology.

- **Organizational rules are modeled as architectural constraints.** Once the architectural description of the system has been made, the resulting design has to respect certain organizational rules. To this end, it is possible to think about these rules as architectural constraints over the configurations of components and connectors (again, expressed via a suitable ADL [15, 21]), as long as the constraints are oriented to capture domain-independent information.

- **Organizational design structures appear embodied in proto-frameworks.** Since we consider organizational structures as architectural patterns, these structures can be similarly mapped to object-oriented constructs.
To evaluate the feasibility of the approach, an architectural MAS model called *Bubble* has been developed. Once specified, this model was translated to its proto-framework counterpart, and later used to implement several frameworks/applications [8].

### 4 The *Bubble* Architectural Model and Its Proto-framework

The *Bubble* architectural model relies on a few basic building blocks, namely: agents, tasks, sensors and events. The model can be described from three different views: structural organization, communications, and behavior (see Figure 3). In the following, we briefly describe each of these views.

- **Structural Organization.** One of the *Bubble*’s design goals was uniform decomposition. By uniform decomposition [1], it is meant the operation of separating a large component into smaller components, but limiting the composition mechanisms to a restricted uniform set. Thus, integration of components and scaling of the system as a whole is achieved by partitioning the system into sub-systems with limited interaction, having besides modifiability and reusability properties. The schema permits to organize agents according to a hierarchy of abstraction levels: agents composed by other agents, which in turn are composed by others, and so on. This organization corresponds with a layered structure, where an agent within a layer (a container) can only be aware of components that reside in the same layer, being mostly unaware of external components. In addition, the same structure is used to handle incoming and outgoing events.

- **Communications.** Events and implicit invocation mechanisms are typical features of architectural styles dominated by communication patterns among independent components [23]. All the communications between entities are achieved by exchanging events. Implicit invocation occurs when an agent executes tasks in reaction to notifications (events) from other agents. These tasks are implicitly activated, because the agents responsible for those notifications do not know if their notifications will cause any reaction on specific destination agents. Although this schema provides a great flexibility to define interaction protocols, interactions may often become difficult to control and generate an excessive event flow in the system. To take this situation into account, the architecture incorporates two complementary mechanisms: hierarchical event handling and sensors attached to the agents. Any agent can be linked to a container agent, and this container is engaged to collect and dispatch incoming events to the sensors registered inside it. Besides, sensors are responsible for the reception and conditional forwarding (filtering) of interesting events to their associated agents.

- **Behavior.** As regards the inner structure of an agent, it comprises a queue of incoming events, an internal state and a set of tasks the agent is able to perform. A task is basically composed by one or more procedures with a set of input and output parameters. Tasks are triggered by predefined conditions, which can be related to the internal state of the agents or incoming events. In this way, the agent’s be-
behavior is conceived as a set of competing tasks, where only one task is active at the same time [10]. When a selected task is executed, it can generate either outgoing events and/or changes affecting the agent’s state. Different tasks can be dynamically assigned to an agent with particular activation strategies. In a way, the behavior of an agent can be seen as a kind of control loop architecture [23].
The process of transforming the *Bubble* model into a proto-framework structure was primarily driven by the need of mapping MAS built from elements of the architectural model to object-oriented constructs. The proto-framework was implemented in Java, adopting a layering technique [11]. That is, it hides to developers all the platform-dependent details as well as the mechanisms for achieving communication between architectural elements, registration and notification of events, activation of tasks, and the like. Additionally, the framework incorporates a number of built-in services regarding threading, distribution and persistency issues. More details about this implementation can be found elsewhere [8]. Figure 4 depicts a simplified class diagram of the *Bubble* proto-framework.

### 4.1 Mapping to MAS Organizations

Intuitively, it can be seen that the *Bubble* model reflects most of the concepts depicted by Figure 1. Each of these concepts and their corresponding *Bubble* mechanisms are summarized in Table 1.

Note that the *Bubble* model prescribes a number of rules regarding the way systems that adhere to this model should be constructed. By capturing these rules along

<table>
<thead>
<tr>
<th>Organizational Abstraction</th>
<th>Bubble Mechanisms</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Agent Organization</td>
<td>Container and composed agents</td>
<td>Organizations can be modeled either as container or composed agents. Composed agents intend to be lightweight containers, as they just define a group abstraction and a set of roles to be played by agents in that group. Both membership and interaction policies for the group may be later specified. On the other hand, container agents provide a stronger structural abstraction, incorporating sensors and dispatching mechanisms within a group.</td>
</tr>
<tr>
<td>Environment</td>
<td>Container agent</td>
<td>This concept can be directly modeled in terms of a container agent. As a consequence, note that <em>Bubble</em> allows a given system to have multiple contexts (environments).</td>
</tr>
<tr>
<td>Interaction Medium</td>
<td>Events and sensors</td>
<td>Communication between agents takes place over the event-based infrastructure of publishers, subscribers and dispatchers supported by the architectural model.</td>
</tr>
<tr>
<td>Interaction Protocol</td>
<td>Container and composed agents, plus sensor</td>
<td>Using the basic inter-agent communication capabilities given by <em>Bubble</em>, more elaborated communication protocols can be implemented at the level of container and/or composed agents. Specific protocol rules may also be enforced by specially-designated agents.</td>
</tr>
<tr>
<td>Sensor</td>
<td>Sensor</td>
<td>This concept is basically the same in the <em>Bubble</em> architecture. A sensor represents an event filter associated to some agent. According to the environment mapping previously defined, the usual context for sensors is generally a container agent. Note that a given sensor should pertain to only one container.</td>
</tr>
<tr>
<td>Effector</td>
<td>Task</td>
<td>This concept corresponds to the concept of task in <em>Bubble</em>. Externally, the execution of a given agent’s task can potentially affect the environment and/or influence other agents.</td>
</tr>
</tbody>
</table>
with the architectural description, we can later provide better guidelines to derive proto-framework structures. In Figure 5, a possible ADL-based formalism to express these rules is outlined. It is loosely based on the Armani-ACME ADL [15, 21], borrowing some predicate for temporal logic and role models from [27].

\[
\begin{align*}
\text{\texttt{\&}}a & \quad \text{is true next} \\
\text{\texttt{\&}}a & \quad \text{is always true} \\
\text{\texttt{\&}}b & \quad \text{is true before } y \text{ is true} \\
\end{align*}
\]

\{\text{Temporal logic connectives}\}

\[
\begin{align*}
\text{\texttt{plays}(ag,r)} & \quad \text{the agent } ag \text{ plays the role } r \\
\text{\texttt{initiate}(r,p)} & \quad \text{the role } r \text{ initiates the protocol } p \\
\text{\texttt{participate}(r,p)} & \quad \text{the role } r \text{ participates in protocol } p \\
\text{\texttt{connected}(c1,c2)} & \quad \text{the component } c1 \text{ is connected to the component } c2 \text{ by at least one connector} \\
\text{\texttt{reachable}(c1,c2)} & \quad \text{the component } c1 \text{ is in the transitive closure of } \text{connected}(c1, *) \\
\text{\texttt{hasType}(c,t)} & \quad \text{the component } c \text{ satisfies the type specification given by } t \\
\end{align*}
\]

\{Zambonelli's predicates for agents, roles and protocols\}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{fig5.png}
\caption{A sample of organizational rules for the \textit{Bubble} model, using an ADL-like notation}
\end{figure}

In the following section, we describe an application framework designed according to the \textit{Bubble} architectural model, which has been implemented by instantiating (and refining) the derived proto-framework.

5 Case-Study: A MAS for Enterprise Quality Management

\textit{InQuality}\(^1\) is an industrial framework, built on top of the proto-framework \textit{Bubble}, for the development of document-based applications in Enterprise Quality Management.

\(^1\) \textit{InQuality}\(^\circ\) is a commercial software developed by Analyte Lab Technology Solutions, Buenos Aires, Argentina.
Systems (EQMS) [6]. From a modeling perspective, *InQuality*-based applications typically consist of a set of related sub-systems organized in a hierarchical fashion, where the components of each sub-system work together to deliver a given functionality. As these systems get bigger, that is to say more components and also more complex interactions appear, the intricate nature of the relationships between components makes it difficult to deal with this kind of systems using just conventional object-oriented techniques. Here, the introduction of agent technology can provide instead a more tractable organizational approach.

5.1 Overview of the *InQuality* Architecture

*InQuality* is designed around a blackboard architectural style combined with events. Each of the components of the system works autonomously, publishing into the blackboard those services it provides and those requirements it needs to satisfy. Figure 6 depicts the main components and interactions of the *InQuality* architecture\(^2\). A short description of these components is given below.

![Architecture of InQuality](image)

**Fig. 6. Architecture of InQuality**

As *InQuality* is web-enabled, components can access the system both using conventional browsers or specific-developed applications using some HTTP-based protocol, for example SOAP (Simple Object Access Protocol). A number of servlets that reside in the web-server environment handle the requests from the clients, and then post these requests into the blackboard. The servlets are also registered with the blackboard to be notified of responses to these requests. Once a given response is received by a servlet, it communicates this message to the corresponding client.

\(^2\) Some specific details of the *InQuality* framework are omitted due to academic-industrial confidentiality agreements.
The Role Manager keeps track of the open sessions, meanwhile verifying the user operations against security rules and permissions. A role model defines the capabilities and the different rights of every type of user. Each session is monitored by an agent, which captures the client requests and tries to satisfy them. The Workflow Manager is responsible for initiating, tracking and ending the workflow instances of the different documents in the server. Because information is uniformly expressed as documents, most of the processes are workflows themselves, operating under the supervision of this manager. The workflow manager additionally notifies roles and participants when they should perform specific activities. The Page Builder constructs the graphical interface for the user when he/she enters the system via a web browser. Documents, frames and interface elements are converted to a representation suitable for the browsers.

As regards the Application Interface Model, this object model acts as a repository for GUI information associated to a particular application, e.g. menus, frames, topic frames, button frames, etc. The Page Builder component uses this information to translate entities from this model to a browser-dependent implementation. The Document Model encapsulates the basic functionality to operate with documents, for example: creation, access, composition, storing, etc. The Security Model is a subsystem of the Document Model, because security is also implemented using documents. This component manages the permissions for the different participants involved in workflow activities. The security rules are used by the Role Manager and the system configuration tools, in order to define and get/set security attributes either on GUIs or domain-specific components. Finally, the Specific Domain Model is the “real” component model for a particular domain, e.g. a Laboratory Information Management System (LIMS) or EQMS. The entities modeled here are often “passive” wrappers for the database elements, which may be complemented with certain “business rules”. We refer to these entities as passive because an associated agent is in charge of managing the autonomous behavior of this kind of components.

5.2 Incorporating Workflow Agents in InQuality

In the case of InQuality, the incorporation of agents should be done as less intrusively as possible. In other words, the applications should continue working normally, unaware of whether the are linked to agents or not. Additionally, the schema should allow enough adaptability to support different “agentified” configurations. To take into account this requirement, a variant of the wrapper pattern [26] was applied. This way, certain domain entities are associated with agents in charge of monitoring them, using the infrastructure provided by the Bubble architectural model. This means, for example, that an agent is continuously watching the behavior of its target component until certain business rules activate some predefined agent’s tasks. When this occurs, the agent will interpret the situation, and potentially execute a number of actions and send particular events to the rest of the agents.
There are basically three types of agents in the design of InQuality: interface agents, document agents and workflow agents. A short description of each agent type and its main functions within the system organization are given in Table 2.

Table 2. Types of agents in InQuality

<table>
<thead>
<tr>
<th>Agent Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interface Agents</td>
<td>These agents are in charge of the interaction with the user through the system graphical interface. When a servlet accepts a given user login (e.g., name and password), there is a Logger agent that receives the corresponding event from its associated sensor, and then activates a specific task that checks if the logging information matches any of the system users. In fact, there is a User agent to represent every user and its roles. If the authentication is correct, the user is logged into the system. Then, a LoggedUser agent is created and associated with the corresponding GUI (see the Page Builder component). If not, an error message is generated. The purpose of the LoggedUser agents is to store session information for the different users operating on the system. These agents exist as long as the sessions are open, handling the interaction of the users with the system according to the capabilities defined by their respective profiles (see the Role Manager component).</td>
</tr>
<tr>
<td>Document Agents</td>
<td>For each type of document in the system, InQuality provides an associated agent, engaged to keep all the documents updated and consistent. These activities are specified by means of script tasks using Jython. To coordinate all these document agents, they are grouped into a container agent called DocumentMonitor. The DocumentMonitor agent has a subscription with the blackboard to receive events every time a new document or version is created, a document is removed, or some document contents are changed. For instance, if a new ISODocument is generated, this document has to be appropriately referenced by the ISOCatalog.</td>
</tr>
<tr>
<td>Workflow Agents</td>
<td>The document’s lifecycle is closely connected with the workflow system. By means of documents, InQuality represents a variety of entities, for example: processes, activities, transitions, or additional data relevant to a given domain. Therefore, the system needs to keep track of the different states these documents go through over time. To do this, there are special workflow agents, as it is described in more detail in the following paragraphs.</td>
</tr>
</tbody>
</table>

As regards workflow management, InQuality has been specially designed in order to automate well-structured repetitive activities that involve processes, documents, roles, resources and activities. This support has been implemented using the abstractions defined by the Bubble architectural model. Generally, a workflow model can be divided into two parts: a static model that corresponds to the process definitions, and a dynamic model (also called runtime model) that effectively implements the execu-

---

3 Because of space limitations, references to implementation details (e.g., class diagrams, code fragments, etc.) are not included, rather we have privileged a more conceptual coverage of the topic.

4 Jython Homepage, http://www.jython.org
tion of the different workflow components. The static model comprises the definition of participants (humans, roles, system resources, departments, etc.), external applications, pools for global variables, activities and transitions, among others. On the other hand, the runtime model represents the instantiation of these components and their corresponding interactions. In particular, it is in this dynamic model where the Bubble model can provide concrete benefits, because the majority of the documents-related processes that reside in the InQuality server can be directly implemented using agent-based workflow instances. Instead of having a centralized workflow engine, every particular workflow process (i.e., a workflow instance) associated with a given document can be modeled as a container agent, and each activity within the workflow process can be modeled as a single agent. This way, scenarios such as the edition-approval-publication of procedure manuals, tracing of client claims, or any other process affecting structured documents can be handled following this organizational model.

At the beginning, each workflow is generated through a Petri-network-based notation using a graphical editor. Then, the editor produces a specification of the network that will be used to configure the specific workflow instances. A workflow-instance agent encapsulates the knowledge related to the structure of the particular workflow and to the creation of the corresponding agents and associated tasks that will implement this workflow. Activity agents listen to the events generated by the execution of previous activities and react when these events arrive. Tasks associated with activity nodes of the workflow will produce messages directed, through the blackboard component, to the specific role or participant agent in charge of executing such an activity. When the event of “end of activity” is captured, the agent for the target activity evaluates the internal rules associated with each action path and triggers certain events, which in turn enable the subsequent activities in the workflow. A possible organizational model for a typical InQuality workflow application is depicted in Figure 7.

![Workflow agents](image-url)
To summarize, the different workflow components are translated to the *Bubble* model on the basis of the following mapping rules:

- The definition of each workflow process is represented by a *WorkflowProcessInstance* container agent.
- The definition of each activity is transformed into a *WorkflowActivityInstance* single agent
- Transition rules are moved to activity sensors, so that both activity agents and sensors emulate the transitions via event mechanisms.
- Communication with external applications is handled by means of special activities encapsulating message invocation.
- Workflow participants (users or roles) are notified of pending work through the execution of different activity’s tasks

### 5.3 Lessons Learned

From the development of *InQuality*, several experiences can be extracted. A first relevant lesson was the importance of having a clear architectural guidance, as the one provided by the *Bubble* model. Generally, the design of EQMS/LIMS frameworks requires more than domain expertise, in the sense that many computational factors that transcend the domain itself should be combined. In *InQuality*, a typical configuration includes: ~100 interface agents (sessions, users, groups and roles), ~30 document agents (document types, version control), and ~1000 workflow agents (processes, activities, roles, transitions, resources, etc.), what make the applications real examples of large-scale MAS. As a second lesson, we realized the importance of communicational aspects. The development team felt more comfortable once a concrete organizational terminology, i.e. agents and architectural constructs, was adopted. This terminology much helped the development team to better transfer their workflow architecture to a running structure, while allocating the required functionality into the computational components prescribed by the proto-framework.

When it comes to technical aspects, we faced a tension between following the core *Bubble* architectural model and taking advance of domain-dependent information to modify the “inherited” proto-framework design [19]. This tradeoff involved the consideration of things like overall performance, concurrency, subsetting of agency features or middleware, among others. For instance, since the building mechanisms included in the *Bubble* architecture are targeted to enforce adaptability issues, the workflow processes of *InQuality* were able to evolve and deal with different variations of a given problem. Actually, using the same architectural ideas, additional development facilities such as visualization, logging and statistics were implemented. But, this implementation turned out to be somehow inefficient regarding performance and event control, due to the internal threading policies and the event dispatching mechanisms supported by the framework. Thus, developers had to carefully control the event flow in the applications, in order to avoid possible overheads. Currently, new re-organizations of the architectural model and the proto-framework, driven by the above observations, are being investigated.
6 Related Work

Considerable research has been done in the area of MAS frameworks and models. In particular, some of the existent approaches have exploited the conception of MAS as a computational organization, for instance, the Madkit platform [16] and the socio-intentional architectural styles proposed by [9]. In the first approach, the authors view MAS organizations as groups of agents engaged to specific roles. Similarly to the Bubble model, Madkit recognizes the need of agent managers (composed and container agents in Bubble) to control the coherence of group structures. Furthermore, all these notions are directly supported by a service-oriented framework. However, the link between the Madkit’s organizational abstractions and the corresponding implementation is quite inflexible, in the sense neither new architectural abstractions (e.g., architectural structures to address specific qualities of the system) nor other organizational concepts (e.g., organizational rules) can be easily incorporated into the core model and later mapped to the Madkit framework.

Regarding the second approach [9], the authors propose a catalog of patterns for MAS that adopt concepts from organization theory and strategic alliance. The patterns are specified using the strategic dependency model of the i* framework, within the context of the Tropos project [17]. Unlike the Madkit approach, organizational concerns are here addressed at architectural levels. As a consequence of following the Tropos’ philosophy, the link between model specifications and implementations admits some degrees of freedom, according to the relevant quality goals of the system.

An arguable aspect of socio-intentional patterns, from a practical design perspective, is that they have not proved yet a clear equivalence to conventional architecture-based techniques, as the ones used by the proto-framework approach, thus socio-intentional styles are not always accepted by practitioners. A shared concern between Tropos and our materialization approach is the traceability between architectural models and their implementation, e.g. by means of object-oriented models. More related to MAS architectural patterns and quality attributes, an application of the ATAM evaluation process to agent systems is reported in [26]. It seems possible to readily apply this body of knowledge to our approach.

When considering “architectural frameworks”, Medvidovic et.al. [19] have developed a family of implementation frameworks for conceptual architectures based on the C2 style. This view is somewhat closer to our proto-framework approach, as it provides a basic architectural framework and then allows developers to address various extra-functional properties required at the application level. Moreover, the C2 model shares many characteristics with the Bubble model, although Bubble design features are more expressive regarding organizational abstractions, namely: flexible definition of interaction protocols, mechanisms for event filtering and dispatching, and dynamic configuration of agent’s behaviors.

Finally, our approach bears many similarities with domain-specific software architectures (DSSA) [25], and to a lesser extent with model-driven architectures (MDA) [24]. A DSSA provides, basically, a software architecture with reference requirements and a domain model, an infrastructure to support it, and a process to instantiate and refine this infrastructure. The reference architecture generally considers tailoring
mechanisms to satisfy the needs of a real application, whether at design-time or at run-time. The infrastructure is often represented by an object-oriented framework. The main difference with the materialization approach is that proto-frameworks are oriented to capture architectural abstractions mostly domain-independent, and their relationship with quality attributes. In the case of MDA, the main idea of this approach is the separation of the functional specification from the implementation of that functionality on a specific platform. Here, the work concentrates on the definition of open standards for system modeling and middleware (e.g., UML profiles), rather than on architectural issues, which seem to be implicit into the models. Furthermore, it differs from our approach in that the application of transformations in MDA refers principally to the conformance between reference models and implementations.

7 Conclusions and Future Work

In this paper, we have reported on our experience with the development of MAS frameworks from architectural principles, and particularly how it concerns organizational issues. Along this line, an approach based on object-oriented architectural materialization has been proposed. The main contribution of this approach is the notion of proto-frameworks, to provide an intermediate stage in the transition from MAS architectures to implementations through object-oriented frameworks. This favors the capture of organizational structures at architectural levels and their further transformation into object models.

As an example of the materialization approach, we have presented an architectural model called **Bubble**, as well as its derived proto-framework. Although most of these features are not unique to this model, their combination provides a remarkable adaptability to support organizations. The **InQuality** framework demonstrates how applications designed according to this architectural model are implemented on top of the proto-framework.

Although proto-frameworks appear as a promising bridge between architectural models and object-oriented models, there are some issues that remain still unanswered, opening opportunities for future work. First is the investigation of techniques to translate the points of variability defined by architectural models to proto-framework structures. Here, we feel that aspect technologies can be an interesting complement for linking architectural elements, quality attributes and framework abstractions. Second is the provision of semi-automated tool support for architecting MAS. A third line of research is the definition of some formalism to specify MAS organizations at the architectural level (the ACME-like language outlined in the paper can be taken as a preliminary proposal). This can lend to MAS the standard benefits of architectural notations, while permitting design-time reasoning about qualities such as performance, reliability, portability, and conformance to architectural styles. Finally, it is hoped that this work stimulates other practitioners to further investigate architecture-driven design techniques for MAS organizations.
References

A Basic Taxonomy for Role Composition

Elke Steegmans¹, Kurt Schelfthout¹, Tom Holvoet¹,
Yolande Berbers¹, Paul Valckenaers², and Bart Saint Germain²

¹ KULeuven Dept. of Computer Science
Celestijnenlaan 200 A, 3001 Heverlee, Belgium
{Elke.Steegmans,Kurt.Schelfthout,Tom.Holvoet,
Yolande.Berbers}@cs.kuleuven.ac.be
² KULeuven Dept. of Mechanical Engineering
Celestijnenlaan 300 B, 3001 Heverlee, Belgium
{Paul.Valckenaers,Bart.SaintGermain}@mech.kuleuven.ac.be

Abstract. Roles are the basic building blocks for defining the behavior
of agents in multi-agent systems. Agents typically perform several roles.
In this paper, we describe analysis and design issues in defining agents
as compositions of roles. In short, specifying the behavior of an agent
entails in essence two issues: which roles are assigned to a particular
agent, and how does an agent select a role in a particular situation.
Both issues can be decided upon either by the designer (i.e. at design
time) or by the agent (i.e. at run-time). This paper describes a basic
taxonomy for role composition based on both issues, and illustrates the
different composition approaches using a case study in the domain of
manufacturing control.

1 Introduction

Roles are a key element in the development of agents and multi-agent systems
[5,20,23]. Few research, however, considers the issue of composing several roles
to describe the behavior of a single agent. In this paper we propose a basic
taxonomy, where three different approaches for role composition are proposed
and where the focus is on combining several individual roles to achieve a working
agent. We propose three different approaches: static, adaptive and dynamic
composition of roles. This taxonomy is achieved by differentiating between the
assignment of roles and the selection of roles. An important distinction is made
between assignment and selection of roles at design time and at run-time. The
composition of roles can be hierarchical, where a separate policy for assignment
and selection between sub-roles can be chosen for each “layer” in the hierarchy.
The proposed approaches encompass important software engineering require-
ments such as flexibility, adaptability and composability. The most important
requirements on which we focus in this paper is reusability and adaptability of
the approaches within different applications. We explain in detail the validation
of our work through practical experience of the three composition approaches in
the product path planning example within the domain of manufacturing control.
In section 2 of this paper we describe some related work into detail. In section 3 we reflect on the work done in function of MAS ORGanizations (MASORG). In section 4 we focus on what composition of roles is and we propose a basic taxonomy for role composition. Section 5 discusses the three different approaches to establish composition of roles in more detail. For each approach a detailed explanation is given, different techniques are proposed, a concrete application in the domain of manufacturing control is used as an example and finally a few words are written on the implications of using that approach. The hierarchy, which is the result of composition of roles, is described in section 6. Finally section 7 concludes this paper and indicates topics of future work.

2 Related Work

A lot of research is already been done on roles in the (multi-)agent community, a non exhaustive list of papers is [1,5,6,7,11,13,14,19,20,24,29]. In this section we describe only a little subset of related work into detail, and in section 7 we categorize these three in the taxonomy we propose in this paper and compare their method with our statechart formalism.

Kendall [20] describes the importance of roles and role models for multi-agent systems. Within her work she describes a way to let applications develop multi-agent systems by reusing existing or defining new role models, walking through the analysis, design and implementation phase of the development process. The main goal of her work is to find a better way to develop role models on the design and implementation level than making use of the Role Object Pattern. She proposes and proves that the use of aspect-oriented programming (AOP) [21], is a new and better way to design and implement role models. In her paper she describes that one of the important future work topics in the domain of roles is the development of a formalism for roles. This formalism is very important for roles and role models to break through in different application domains and to let the applications develop roles and role models in an easy and intuitive way.

In [1], Agent UML (AUM) is proposed, which is the agent-oriented version of the object-oriented Unified Modeling Language (UML is a standard for doing object-oriented analysis and design). The need of dynamic role assignment is the topic of a recent paper of Odell, Parunak e.a. [24], in which they propose how dynamic role assignment can be specified in AUM. Dynamic role assignment consists of two categories of “role-changing” operations: dynamic classification and dynamic activation. For specifying these “role-changing” operations they propose extensions to the interaction diagrams of UML.

Within the GAIA methodology [29] a roles model on the analysis level is proposed to define and model roles for agents. Each role is described in role schemata, in which a description, the protocols and actions, the permissions and the responsibilities of the specific role are defined. The role model is comprised of a set of role schemata, one for each role in the system. The GAIA methodology is appropriate for applications with the following characteristics: the abilities of the agent are static, they can not change at run-time. The organization structure is static, meaning that the inter-agent relationships do not change at run-time.
3 Multi-agent System Organizations

In MAS ORGanizations (MASORG), a multi-agent system is an organization consisting of several kind of agents which exhibit particular roles in a particular environment. The proposed methodology of MASORG supports each of these abstractions equally, and is depicted abstractly as an equation: Organization = Environment + Roles [16]; an organization is a combination of an environment and roles that are consistent with this environment.

A role is a partial specification of agent behavior: it isolates the part of an agent that is relevant to a behavior. An agent that can only play one role is completely specified by that role. When it has multiple behaviors, that are each specified in a role, the composition of those roles specifies the agent behavior. Roles represent more than a set of interface operations, but rather incorporate their own autonomous behavior. An agent may exhibit several roles at a time, and several agents may play an identical role in a multi-agent system. The existence of a role depends on the existence of the agent that performs the role: an agent and its roles have one identity. This is analogous to the definition of a role in OO-terms [26]. Examples of roles are: the searching role, the manager role, the supplier role, the answerer role, . . .

In this paper, we report on further work that focuses on role composition. In a first report on MASORG [17], a specification method or formalism was proposed focusing on two important aspects: the specification of one role and the reuse of roles within different applications. To accomplish this we distinguish between two views in MASORG: the generic view and the application view.

Generic View in MASORG: In the generic view of MASORG the application-independent behavior of roles is modeled using statecharts. Such a statechart represents the role an agent plays in the organization; it describes the behavior of the agent in an abstract way. This results in a specification method for defining abstract roles, offered to be reused within different applications, so that the application developers do not need to describe the same behavior over and over again.

Application View in MASORG: In the application view, application developers are able to reuse abstract roles and to indicate where/when/what application-specific functionality needs to be performed by the agent that exhibits this role. For this purpose we provide five new mechanisms to extend the statechart of an abstract role: preconditions for actions, pre-actions, post-actions, pre-state-actions and post-state-actions.

4 Composition of Roles

The behavior of an agent exhibiting just one role is straightforward: it behaves as described by its role. An agent consisting of several roles on the other hand poses significant extra challenges: roles in general can be dependent in arbitrary ways, and the decision to execute a certain role in a certain situation is non-trivial. We propose a number of approaches for composing roles. All the roles that are
used within the different composition approaches are all specified making use of the same formalism and role composition idea.

4.1 Composition Is Assignment and Selection

The behavior of an agent first depends on the assignment of a set of roles to an agent, indicating that the agent is capable of performing these roles at a particular point in time. The fact that an agent is assigned several roles is useful when a number of basic behaviors are offered as basic roles. These basic roles are very generic, meaning that they are useful in different applications or situations. Assigning several of those basic roles to one agent gives that agent more functionality and results in a more specific behavior of that agent. Another reason for assigning several roles to one agent is giving the designer the possibility to develop agents with a broad range of behavior/functionality, agents that are able to do different things and are able to react in different circumstances. Second in the development of an agent, the designer needs to decide upon the relationship between the different roles assigned to the agent - in other words, the selection criteria, that establish in which situation which role will become active.

The designer of the multi-agent system can choose to define each of these two composition phases either at design time or at run-time. At design time means that the designer defines the behavior of the agents before they start running. At run-time on the contrary means that the designer of the multi-agent system wants to add or remove roles, and let the agent adapt the selection criteria, while it is running. We elaborate in the following paragraph.

4.2 A Basic Taxonomy for Role Composition

In Table 1 the different approaches for composing roles are depicted. A first approach is called static composition, where both the assignment and the selection phase are handled at design time. Adaptive composition allows the assignment of roles to be handled at design time, but the selection of roles at run-time. The last approach, called dynamic composition, allows to handle both the assignment and the selection of roles at run-time. The approach not depicted in Table 1, would allow the designer to handle the assignment of roles at run-time and the selection of roles at design time. This approach however appears useless because it would allow defining and assigning new roles at run-time, while the agent is not able to execute these new roles because the selection criteria can not be changed at run-time. The three approaches depicted in Table 1 are explained in more detail in section 5.

<table>
<thead>
<tr>
<th>Approach</th>
<th>Assignment</th>
<th>Selection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static</td>
<td>at design time</td>
<td>at design time</td>
</tr>
<tr>
<td>Adaptive</td>
<td>at design time</td>
<td>at run-time</td>
</tr>
<tr>
<td>Dynamic</td>
<td>at run-time</td>
<td>at run-time</td>
</tr>
</tbody>
</table>
Which approach to use in a particular situation depends on the target application. Each approach has its own characteristics, which may or may not be sufficient for the targeted application: Static composition builds very predictable agents, but trades off in adaptivity. Predictability is an issue in many high-risk environments, such as airplane control. While adaptive composition builds less predictable, but more adaptive agents. Although the predictability on the individual level decreases, the system as a whole is much more responsive to changes. This can be useful in domains such as manufacturing control, where responsiveness to changing demands is more important than an “exact” solution to the constantly changing optimization problem. Dynamic composition allows for a very extensible agent. The designer can now intervene while the agents are running, and add or remove roles from the agent. This may be necessary for agent systems that are undesirable to shut down, and new features or bug fixes need to be done at run-time.

4.3 Application: Manufacturing Control

In section 5 we discuss the three approaches in more detail and we elaborate on applying them in a concrete example. We develop an example in the manufacturing control domain based on the behavior of ants. The behavior of ants has already been proven to be useful in manufacturing control [10,28]. In this example the product path planning task is developed: when a new order for a certain product is given, agents need to detect a possible path through the plant to manufacture that product. The agents need to find a path through the plant, consisting of a graph of resources (e.g. as in Fig. 2), for building the product. Following this path, consisting of a sequence of resources, will result in performing the operations in that specific order needed to build the product. We describe three case studies in function of this example, one for each composition approach. However, the roles described here are generic roles: they are also applied in routing problems in active networks [9,10].

5 Three Approaches for Composing Roles

The three approaches from Table 1 will now be discussed in more detail. For each of the approaches, we discuss some possible techniques and a concrete example in the domain of manufacturing control. We evaluate the benefits, drawbacks and requirements of each approach.

5.1 Static Composition

If the behavior of the agents is entirely known at design time, the designer can mould the behavior in an algorithm. Such an algorithm describes the behavior of an agent as a number of steps the agent executes in a particular predefined way. The responsibility of defining the behavior for the agents is entirely for the designer, the agents can not make any decisions of their own. The two phases are:
– Role Assignment Phase at Design Time: the set of roles is defined at design time, the agents are created and get assigned their roles.
– Role Selection Phase at Design Time: the selection criteria for each role are defined at design time indicating in which specific situation which of the agent’s roles need to become active.

Techniques. Building further on the use of statecharts to model roles, there exist a number of techniques (analogous to techniques in structured programming) for defining relations between roles at design time:

– Single Execution: the single execution allows to define that an agent executes one of its roles at one moment. For example there exists an agent with three roles assigned: A, B and C. The designer can decide to relate these roles by using an “exclusive-or” relationship between them, meaning that the agent is in role A or in role B or in role C at one moment in time. This is described by using an XOR state in statechart terms [15]. A number of techniques can be used to define a relationship between or compose such exclusive-or related roles:
  • Sequence: the sequence relation allows to define the sequence in which a set of roles belonging to an agent is executed. The sequence relation allows the designer to define a kind of ordering relation between the roles of an agent. For example when an agent is assigned three roles, the designer can select the roles using the sequence relation indicating that the agent first executes role A, then role B and finally role C, in this specific order. The next two relations are special kind of the sequence relation:
    * Iteration: the iteration relation allows to indicate that the same role needs to be executed repeatedly a number of times. For example, an agent with two roles can be defined by the following selection of roles: first execute three times role A repeatedly and then two times role B repeatedly.
    * Conditional Sequence: the conditional sequence adds a condition to the role that an agent performs next. For example, a designer can specify that, an agent that executes role X at some point in time, will execute role Y if condition y holds, and role Z if condition z evaluates to true.
– Parallel Execution: the parallel execution relation allows to define that an agent executes two or more roles in parallel. For example there exists an agent with two roles: role A and role B. The designer can decide to relate these roles by using an “and” relationship between them, meaning that when the agent is in a certain state of role A and simultaneously in a certain state of B, the agent is executing in parallel the behavior of role A and role B. In this situation the agent can perform the next action from role A or the next action from role B. This is called orthogonality in statechart terms and is described by using an AND state [15].
Manufacturing Control Case Study Using Static Composition. In this section we discuss a case study in function of the manufacturing example as described in 4.3 using the static composition approach. In this first case study the designer starts from the following situation: a non-running factory where orders for different products arrive. The designer knows at design time that the agents he is developing need to walk through the plant, looking for a path through it to manufacture their product. The designer could take the following decisions for the assignment and the selection of roles in this case study:

![Diagram of Static Composition]

**Fig. 1. An Example of Static Composition**

- **Role Assignment Phase:** The agents need to execute the following two roles to establish their goal: a scouting role and a marking role. The scouting role will lead the agent to detect a possible path, while the marking role will mark this path as a possible path to manufacture their product. Each agent can execute both roles, and gets assigned both of the roles, meaning that they are all able to execute the behavior of the two existing roles.
- **Role Selection Phase:** In this example it is obvious that the agents need to perform the scouting role first and afterwards, if the agent has detected a path for the manufacturing of its product in the plant, the marking role. The path is detected at the moment that the agent is at an end node of the
plant and the sequence of operations to perform manufacturing the product is completed (a little remark to make here is that we assume that there exists at least one path through the plant as we do not want to make the example enormously complex therefore there is no checking on the completion of the sequence of operations). At this point, the agent switches from its scouting to its marking role and the marking role will lay a trail back to its starting point indicating that there is a possibility to manufacture its product through this path. This selection scenario with one conditional sequence is depicted in Figure 1, in which the semantics of Harel’s statecharts is used [15]. However, for a conditional action we use a separate box for both the precondition (light gray box) and the action (white box), while this would be written “action(condition)” in conventional notation. This to improve readability since we extended the statechart formalism with extra features (see 3). Being in its scouting role, the agent can switch to its marking role under the condition that he has detected an end node in the plant, depicted in the figure by the white box above the “start to return” action (also depicted by a white box in the figure) with the following condition: if (! neighbors). The agent is at one point in time always executing or the scouting role or the marking role, depicted in the figure by using the XOR relationship between the roles.

![Diagram](image.png)

**Fig. 2.** Situation sketch of the plant in the first case study

In Figure 2 the result of an agent executing the behavior described above is depicted. The agent scouted first through the plant along the resources A, B, C, E and G. Arriving in the end resource G, the agent marked this path by switching to its marking role because it had detected a path along which it could manufacture its product.

**Implications.** When there is need for dependent roles and the behavior of the agent is known at design time, this selection mechanism is very useful. Because it results in a high-coupling between the roles, the composed roles define a specific behavior, which is less generic. The composition of such roles within this approach can start with using a number of little, fine-grained and generic basic roles and composing them into a more specific, fixed and less generic behavior. Thus it is important to offer a set of very basic and fine-grained roles, which can be reused by a large number of applications and can be composed to a specific behavior at design time.
5.2 Adaptive Composition

If agents need to be more flexible and responsive, it is possible to make the role selection mechanism dynamic. In the previous approach, the decision to move from one role to another, was fixed by the designer (i.e. at design time). In the approach described in this paragraph, this responsibility is shifted to the agent. The two composition phases are as follows:

- Role Assignment Phase at Design Time: This is identical to the previous approach.
- Role Selection Phase at Run-Time: For some applications, the designer would like the agents to be adaptive and responsive to changes in their environment. The agent’s decision is based on cues from the designer, but above all on an assessment of current observations, and possibly past experiences. In other words, the final responsibility for the decision lies with the agent, and thus at run-time.

We therefore add a mechanism that is able to observe the agent’s environment, and, based on past experiences and on cues from the designer, decide which of its roles the agent will execute. We elaborate on some possible techniques in the following paragraph.

Techniques. It is important to stress that there are various approaches to an adaptive role selection mechanism, since this problem is closely related to the more general problem of task allocation. Some existing mechanisms are learning based approaches, where each agent learns when to execute each role. This can be done using feedback mechanism such as reinforcement learning [18]. Another approach is negotiation, where agents negotiate with other agents what role to execute using a certain protocol [22]. An example of this would be the Contract Net protocol [8]. A third approach would be a behavior-based approach, originating from robotics. In this approach, the agent consists of a predefined set of behaviors. At run-time, the agent can decide which of these behaviors will become active. An example of such an architecture is the subsumption architecture [4].

As an important technique for dynamic selection, we consider a stimulus-threshold algorithm. This mechanism is used as a model for real ant behavior [3]. The basic idea is to associate both a stimulus and a threshold with each role, a full explanation is described in [25]. The stimulus for a certain role increases when its execution is more urgent or more important. Whenever a stimulus exceeds its threshold, the corresponding role is executed (ties can be broken in various ways - one would be to pick the role whose stimulus exceeds its threshold the most).

The thresholds can be set by the designer, and are thus an indication of the importance of the role. On the other hand, no matter how high the threshold, if the stimulus for a role becomes sufficiently high (the role becomes urgent), it will be executed anyway. In this way, the designer can still control the behavior of the agent by setting the thresholds, while the agent on the other hand remains
responsive to its environment. Several variants of this scheme exist, e.g. varying threshold models [25], hierarchical models, ... This model has two main benefits. First, it is scalable to many agents and many roles, since no negotiation, message passing or state transfer between agents is involved. Second, the responsiveness of the model to changing environments is very good, as argued in [2].

Having a number of possible techniques for adaptive role selection, the statechart formalism allows the designer to extend the static approach with the adaptive role selection mechanism. In this approach, the AND state of statecharts is very useful because of its orthogonal property. For example when the designer chooses the stimulus-threshold technique, he needs to add this as the role selection mechanism to the set of roles of the agent. Within this role selection mechanism, the designer sets the threshold values for each of the agent’s roles as well as defines how the stimuli coming from the environment are calculated. The main difference with the static composition approach is this additional role selection mechanism, which allows an agent to control its own decisions. This role selection mechanism is itself represented as a role, and can thus be considered as a meta-level role. In the next paragraph an example of a role selection mechanism is described in more detail.

**Manufacturing Control Case Study Using Adaptive Composition.** In this section we discuss a case study in function of the manufacturing example like described in 4.3 using the adaptive composition approach. In this second case study the designer starts from the following situation: a running factory where a new order for a particular product arrives while the resources are already up and running to manufacture other products from earlier orders. The designer develops agents that are able to detect a path, but that are also able to help for example other agents transporting products to the stock room when in an ending resource an enormous queue of finished products has arrived. The designer develops at design time new agents for that new order. The designer could take the following decisions for the assignment and the selection of roles in this case study:

- **Role Assignment Phase:** The set of roles needed are the path detecting role and the transporting role. The path detecting role is the super role of the composition of the scouting role and the marking role like depicted in Figure 1.

- **Role Selection Phase:** In this case it is not completely obvious at design time in which situation the agent needs to switch to its transporting role. Therefore it is better to choose for the selection of roles at run-time, resulting in the fact that the agent can decide itself when it switches roles. The agent can decide to switch to its transporting role when it is searching for a path, and arrives in its end resource where an enormous queue of finished products is waiting to be transported to the stock room. In this situation it is better to first help transporting those finished products to the stock room, so that the resource’s load is diminished. This in turn reduces the operation time the agent is calculating for the manufacturing of its product.
To allow the selection to happen at run-time, the designer decides to use a stimulus-threshold algorithm. In this case he adds a role selection mechanism like depicted in Figure 3 (note that not all the details are depicted in this figure, just the information needed to clarify the example). The role selection mechanism, depicted by the dotted rounded square, is added to the set of roles and placed in parallel with it. Once the agent starts running, he becomes active in the role selection meta-level role and in the “path detecting&transporting” role, more specifically in the “Initialize Thresholds” state. In the path detecting role, this is indicated by the default arrow (arrows starting with a circle and ending in an arrow). As already mentioned above, the role selection meta-level role and the “path detecting&transporting” role are connected by an AND relationship (depicted in the figure by the vertical dotted line), meaning that the agent is active in both of them in parallel. Being active in the role selection meta-level role means that first of all the thresholds for each of the roles (the path detecting and the transporting role) are initialized, in the “Initialize Thresholds” state. The next state to become active is the “Calculate Stimuli and Thresholds” state, where the stimuli and threshold values are compared and updated, as described in the used stimulus-threshold algorithm. Then the agent decides whether or not he needs to change from its active role to another role. When the agent
decides that it is not necessary to change to another role (depicted in the figure by the result of the C state, which is a conditional state), it stays in the “Calculate Stimuli and Thresholds” state and repeats its behavior when new stimuli values are given from the environment. When the agent on the contrary decides to change to another role, it selects and executes the new role. For example, the agent was executing the path detecting role, but because the resource is too overloaded to get an effective calculation of the operation time (given as stimuli from the environment), the agent needs to change to its transporting role. In this case, the agent performs the “Transporting Role selected” action ending in the “Change to Transporting Role” state, performing the “start transporting” action in parallel in the role selection state and in the “path detecting&transporting” role. This puts the agent in the “Calculate Stimuli and Threshold” state of the role selection state and in the transporting role of the “path detecting&transporting” role. Thus in this approach the ultimate decision as to what role to execute lies with the agent, at run-time, and depends on the current stimuli of the environment and the thresholds.

![Diagram of role composition](image)

**Fig. 4.** Situation sketch of the plant in the second case study

The example is depicted in Figure 4. The status of a resource when it is overloaded is depicted in the figure by grey tints, an example is the end resource G which is colored darkly grey indicating that it is highly overloaded. The agent executing its scouting role of its path detecting role arrives in the resource G detecting that this resource is very much overloaded, so the agent switches to its transporting role helping to bring the finished products to the stock room. Once the situation at the resource G is stable again, the agent switches back to its path detecting role in the state it has left it the last time it was executing this role (depicted in Figure 3 with the H in the path detecting role (the history symbol of the statechart notation [15])). Thus the agent detects that the resource G was the ending resource of its path and it switches from its scouting role to its marking role in the path detecting role, executing the marking role like depicted in the figure.

**Implications.** Introducing a run-time selection mechanism results in a design trade-off between the following two forces: loosely coupled roles and coherent behavior. *Loosely coupled roles:* the composition process is straightforward when
there are few dependencies between the roles. No new dependencies or relations are introduced between the roles, as opposed to the static selection approach in the previous paragraph. This promotes the decoupling of individual roles, and allows more reuse. *Coherent behavior*: on the other hand, some roles are inherently very dependent on other roles. It may be difficult to capture some dependencies such as those described in the previous paragraph between the scouting, following and transporting role, using a very dynamic selection mechanism. We are not claiming that this is impossible, but rather that the overhead that will necessarily be involved may be too much.

## 5.3 Dynamic Composition

This approach is the most complicated of the three. However, the usefulness is incontestable. If a designer wants to change or add roles to an agent, without being able to actually stop the agent, recompile it and start it again, it is necessary to both *assign* and *select* new roles at run-time. Such a situation can arise if an agent is very important for the system to work correctly, but the agent needs extra behavior, or its current behavior is buggy. Due to external constraints, the system cannot be shut down. In such a case, it is useful if the designer has the ability to use mechanisms for adding and deleting roles (and recomposing roles) at run-time.

**Techniques.** As to the role selection mechanism, it is obvious that this needs to select roles at run-time (like described in the adaptive composition approach), since new roles can be added at run-time. However, the mechanism to add new roles can vary. We discuss and propose two possible techniques:

- Designer Control: In this case, the application designer picks one or several agents manually, and adds or removes a role. The agents themselves do not "know" that a new role is added.

- Agent Control: In this case, the agent decides on his own whether or not to add a new role. The designer can for example put a role in the agent’s environment, where agents can pick it up and add to their set of roles. In this case, the designer does not need to know explicitly which agent has which role - in essence, agents could decide for themselves to exchange, add or remove roles. The composition method is now fully dynamic: the designer only knows that there is a set of roles, that may expand or contract at run-time, and a set of agents. He knows nothing about the mapping between the two.

Adding a role at run-time implicates that not only the description of the behavior of the role needs to be defined before dropping it in the environment or adding it to an agent, but it is also important that the selection criteria for this role is known so that the agent knows when he needs to execute this new role. For this approach, a combination of the previous two statechart formalisms in the static and adaptive approaches can be used.
Manufacturing Control Case Study Using Dynamic Composition. In this section we discuss a case study in function of the manufacturing example like described in 4.3 using the dynamic composition approach. In this third and last case study the designer starts from the situation described in the previous case study: the factory is up and running and an agent is starting to walk back to its starting resource. However the following situation occurs: resource C is a painting machine and is running short on paint. For this reason it offers a filling up role to agents passing by it, to get itself filled up again with paint. This is an example of the environment offering a new role, so that running agents can pick it up and add that new role to their set of roles at run-time. This situation is depicted in Figure 5. The agent marking its detected path, arrives at the resource C, which is colored black indicating that it is running short on paint and offering a filling up role to agents passing by. When the agent arrives at resource C, it detects that this resource is in trouble and picks up the filling role, executes it by going to the paint room and bringing paint from it to the resource. Arrived back in resource C, it starts executing again its marking role which it was executing before. The agent makes its decisions based on a stimulus-threshold algorithm. The whole case study is handled at run-time.

**Fig. 5.** Situation sketch of the plant in the third case study

**Implications.** In this last approach, the composition of roles at last becomes fully dynamic. The two different possibilities, designer control and agent control, trade off agent simplicity for more dynamism. In the designer control scenario, there is still a great deal of complexity for the designer: he must know which agents need which new role. It implies that each agent can be uniquely identified and “taken out” of the system, which may not be desirable in all cases (e.g. mobile agents). The agent control mechanism is much more flexible, and allows the definition of new roles. It seems clear that dynamic composition will entail a significant amount of overhead compared to static composition. When the designer chooses between different approaches, it is important that he considers the trade-off between dynamism and efficiency.
6 Hierarchical Composition of Roles

Composing roles results in defining the entire behavior of the agent. In this paper, we propose a way to establish the composition of those roles by iterating a number of times over the assignment and selection phase. This results in a number of layers, building up a hierarchy.

Like already mentioned above, the hierarchy is built up in a number of layers. At the bottom layer of the hierarchy a number of basic and highly generic roles are composed, while at the top huge, very specific and less generic roles are composed. Building up these layers can be done by using the various approaches described above. A hierarchical scheme is necessary for two main reasons: reuse and modularity. 

**Reuse**: more fine-grained building blocks offer more possibilities for reuse. The ability to hierarchically compose roles offers the ability to reuse applications both using small blocks and using big blocks, thus increasing the flexibility of the composition mechanism greatly.

**Modularity**: building-block roles that are fine-grained, but can be re-composed into higher layer roles, that can again be composed to yet higher layer roles, are an essential mechanism for abstraction of these components (roles). Such a mechanism is necessary for large scale systems where it is necessary to keep the design manageable on each layer, while still being able to “zoom in” and expose the details on an underlying layer.

![Diagram](attachment:image.png)

**Fig. 6.** Hierarchy of the “product path planning” example

Throughout section 5 a concrete example of building up such a hierarchy is developed and the hierarchy itself is depicted in Figure 6. The static composition of the first case study of the manufacturing control example is depicted in the rectangle scattered over layer one and layer two of the hierarchy. In layer one
the basic roles, the scouting and marking role, are reflected. While a part of layer two reflects the result of determining the selection between those two roles at design time into a super role, called the path detecting role, like described above. The second case study is depicted in the figure by the detection of the transporting role at design time in layer two of the hierarchy. While the selection at run-time between the two roles of layer two results in a new super role in layer three, called the planning role. And finally, the third and last case study used in the manufacturing example is depicted in the figure by adding of the filling up role at run-time to the agent(s), reflected in the figure in layer three. In this case the selection needs to be done at run-time, resulting in a new super role at layer four, the path planning role. This one is the super role of the hierarchy of this concrete example and is built up making use of the three different approaches proposed in this paper. This illustrates that different composition methods can occur at different levels in the hierarchy.

7 Conclusion and Future Work

The focus of this paper is on role composition, where the question is how to combine individual roles to achieve a working agent. We described a taxonomy of different role composition approaches, based on whether aspects of composition happen at design time or at run-time. We then described the three resulting approaches in more detail, and presented an application to motivate the usefulness, adaptability and reusability of each approach. In that regard, this report is a study on the subject of inter-role composition. A lot of work has been done on roles (static semantic of roles and interactions between roles) like described in section 2, but few of them propose a formalism for defining and composing roles in a fine-grained way. Defining interactions between roles always is accomplished by the designer by defining a relationship between role A and role B. But it is not possible to “zoom in” in the roles A and B and define the relationship in a more fine-grained way, by for instance defining that an agent can only switch from role A to role B after having executed action a of role A. This can be accomplished using MASORG and is one of the strengths of the statechart formalism combined with the composition approaches.

We developed a working Agent-Roles Engine that allows the composition of roles like described above. In Figure 7 a screenshot of the Agent-Roles Engine is depicted, the example shown is part of the path detecting role described and used in the static composition approach and also depicted in Figure 1. We are now using this Agent-Roles Engine in application domains such as manufacturing control and active networks. It is not only important that a formalism for defining and composing roles is developed, but it is also very important and necessary that we also describe it in a formal way. Therefore we have started to work on a formal model to specify agents and roles like proposed in the statechart formalism.

In this paper we propose a basic taxonomy for role composition and extend our statechart formalism, which already allowed to specify roles and add
application-specific functionality to roles, with the ability to do composition of roles. As one of the conclusions of this paper, we situate/categorize the work of others (the three described into detail in section 2) within this taxonomy and/or relate it to our formalism.

As part of the methodology within MASORG, we propose a formalism for roles and composition of roles based on statecharts. With this formalism we propose an answer to the open and important future direction Kendall described in her paper. One of the focuses of our formalism is on reusing roles in different applications by adding application-specific functionality in an easy way and based on the AOP [21] approach. Within the work of Kendall the roles are identified and composed at design time, therefore we categorize her work under the static composition approach of our taxonomy. Odell, Parunak e.a. are working on dynamic role assignment as described in [24], the dynamic role assignment was one of the phases of the dynamic composition approach of our taxonomy. The statechart formalism allows a more fine-grained definition of roles and interaction between roles than the proposed extensions to interaction diagrams in the AUML approach. In our opinion both approaches are complementary to each other, but further investigation needs to be done to reach a definite conclusion. The GAIA methodology focuses on static agents and organizations, therefore this methodology is categorized under the static composition approach of the proposed taxonomy.

In the long term, it is important to tackle a third composition layer: the composition of the multi-agent system as a whole. In this case, the agents themselves are the building blocks, and they need to coordinate their roles composing organizations of agents in such a way that the MAS behavior as a whole is what
the designer wants. Another important future work topic is to handle conflicts that arise while composing roles, such as Kendall also states as an important issue to research in the future.

**Acknowledgment**

This paper presents results from research sponsored by the research council of the K.U.Leuven. The results have been obtained in the Concerted Research Action on Agents for Coordination and Control - AgCo2 project.

**References**

Object-Oriented Modeling Approaches
to Agent-Based Workflow Services

M. Brian Blake¹ and Hassan Gomaa²

¹ Department of Computer Science, Georgetown University, Washington, DC 20057, USA
blakeb@cs.georgetown.edu

² Department of Information and Software Engineering, George Mason University
Fairfax, Virginia 22030, USA
hgomaa@gmu.edu

Abstract. With the increasing popularity of component-based services and semantic web services, the idea of specification-driven service composition is becoming a reality. With the distribution of these autonomous services, a realizable goal will be the transformation of the Internet into a universal service repository. In such an environment, intelligent agents can play a significant role in configuring and enacting the workflow composition of the atomic distributed services to create entirely new higher-level services. In this work, there is a large-scale agent-based architecture to support such a distributed service environment. Furthermore, we introduce an object-oriented modeling and software engineering approach towards the development, configuration, and operational control of the agents that manage processes in this cross-organizational workflow environment.

Keywords: Agent architectures, Software Process, Workflow, UML, object-oriented modeling

1 Introduction

On-line businesses are beginning to adopt a developmental paradigm where high-level component-based services and semantic web services [20] are becoming sufficiently modular and autonomous to be capable of fulfilling the requirements of other businesses. We use the term "services-based cross-organizational workflow (SCW)" to describe the workflow interaction that occurs when one business incorporates the services of another within its own processes (also described as business-to-business (B2B) [3]). This term is also associated with the idea of a third-party organization that composes the services of multiple businesses (also described as virtual enterprise [9]). Though there are many other related projects that define cross-organizational workflow, we further distinguish our work by defining an architecture that uses the autonomy of agent technologies.

In our work, the SCW environment incorporates the interoperability of general services, using the web services-oriented paradigm or using local invocation. In particular, this environment contains general services, web-based services, component-
oriented services, or invocation-based services. In Figure 1, an example is given of a SCW environment for multiple travel-related businesses. The initiating business is the travel agency company. The Travel Agency has internal services for managing customers’ accounts and credit card numbers. However, the travel agency uses other third-party vendors to realize the hotel reservation and car rental reservations. The Hotel Reservation and Car Rental companies register their offerings as web services in a distributed registry, such as a UDDI registry [18]. The Travel Agency uses these registry services as a part of its internal workflow. In addition, the Travel Agency has a partnership with an on-line publishing company that publishes the finalized itineraries. In this case, the travel agency has a static connection with the partner organization and is able to access services directly over a shared network connection.

In the event that the Travel Agency has automated services to collect customer requests, local developers may desire to set up a SCW environment to automatically respond to these customer requests. To support this scenario and possibly other business-oriented routines, there is the need for a framework that supports some process and methodology for workflow-oriented service specification. The workflow developers should be able to specify the process sequence of the local and distributed services. In addition, the message exchange must be specified. The specification methodology must contain support for non-functional concerns. The combination of all of these specifications should be adequate to configure the SCW framework.

Agents in this environment are characterized as having weak agency [14], which means that they have proactive and reactive characteristics as well as knowledge of their environment. In this work, we describe how an agent-based architecture can be used for the realization of this framework. In addition, we suggest the use of standard software engineering processes and languages for the specification of the aforementioned workflow processes and control. In Section 2, there is a discussion of related research in this area and, in Section 3, we detail an approach toward the realization of the SCW environment. In Section 4, we introduce a new approach to modeling...
agents for this environment. Next, there is a discussion of the software developmental process for developing agent systems in this context. Finally, we discuss our experiences using these approaches.

2 Related Research

There are other projects that also investigate the workflow composition of the services. Casati [6] in the eFlow environment uses flowchart approaches to specify the workflow composition of services. Then, both the workflow and individual services are specified in an XML format. Benatallah [2][22] conducts research that uses Unified Modeling Language (UML) [11] statecharts and agent-oriented methods for declarative peer-to-peer service composition. The research does not specifically claim to handle workflow, but the process-oriented specification is related to workflow. In addition, there is an architecture and object-oriented method for describing the process enactment. Benatallah’s work is the most similar to our research and a comparison of the two approaches is given in Section 8. Previous research [12] has also described different approaches for inter-agent communication in multi-agent systems.

There are also projects that concentrate specifically on agent theories for workflow enactment of services. Helal [9] uses an agent architecture for workflow enactment with consideration of services. Chen and Griss [7] also consider the use of agents for workflow with semi-structured specification languages. Finally, Singh [17] discusses the workflow composition of services as a community of services. The major emphasis in this work is an approach to the discovery of services. In the following sections, the WARP approach will be defined and compared to the aforementioned related projects. In the Discussion section, the innovations of this approach are discussed in detail.

3 Overview of the WARP Approach

The Workflow Automation for Agent-Based Reflective Processes (WARP) is an approach for realizing the SCW environment, which builds on previous research. The WARP architecture is divided into two layers, the application coordination layer and the automated configuration layer. The application coordination layer is the level in which the workflow instances are instantiated and the actual workflow execution occurs. The application coordination layer consists of two agents, the Role Manager Agent (RMA) and the Workflow Manager Agent (WMA). The RMAs have knowledge of a specific workflow role. The WMA has knowledge of the workflow policy and applicable roles. When a new process is configured, the workflow policy is saved in a centralized database, which is used as the agents’ knowledge base. The RMA plays a role in the workflow execution by fulfilling one or more services as defined by the workflow policy in the centralized database.

The RMA registers for workflow step-level events in a centralized event based on its predefined role. When an initiation event is written into the event server, the
RMA is notified. Subsequently, based on its localized knowledge of services and its workflow role, the RMA invokes the correct service. The WMA has similar functionality, but instead registers for overall workflow level events (i.e. workflow initiation and nonfunctional concerns). The WMA does not control the workflow execution, but in some cases it adds events to bring about nonfunctional changes to the execution of the entire workflow.

At the automated configuration layer, agents accept new process specifications and deploy application coordination layer agents with the new corresponding policy. This layer consists of the Site Manager Agents (SMA) and the Global Workflow Manager Agent (GWMA). The GWMA accepts workflow representations/specifications from a workflow designer as input. The SMAs discover available services and provides service representations to the GWMAs. This discovery can occur reflectively for local services or from a UDDI registry for distributed services. The GWMAs accept both of these inputs and writes the workflow policy to the centralized database. The GWMA then configures and deploys WMAs to play certain specialized roles. At the completion of workflow-level configuration, the SMA configures and deploys RMAs to play each of the roles specified in the workflow database. A general view of the WARP architecture is shown in Figure 2.

![The WARP Architecture](image)

**Fig. 2.** The WARP Architecture.

### 4 Agent-Based SCW Software Process

Software design and configuration in the WARP environment consists of five main steps as illustrated in Figure 3. In the first step, Site Manager Agents are deployed locally or with access to distributed services that are required for the cross-organizational workflow composition. This discovery can occur on services in UDDI
registry or also in local component-based service registries. These SMAs search for relevant services, and, in the second step, save the service characteristics in the service-oriented data model. Also in the second step, with help from the Global Workflow Manager Agent, these service characteristics are captured in WARP models. In the third step, a workflow designer accesses the available services as Service Representation Views (to be discussed in greater detail in Section 5). The workflow designer then creates the cross-organizational process models. Once the process modeling is complete, in the fourth step, the GWMA captures the WARP models and extracts the raw information. This raw process information is stored in the process-oriented data model. Consequently, an integrated data model of both service and process data models is ready for agent access. In the final step, application-layer agents (Workflow Manager Agents and Role Manager Agents) access the integrated data model and configured themselves for workflow enactment in the SCW environment. The focus of this paper is on the modeling process; however in other work the agent self-configuration operations are presented in detail [4].

In the following section, each of the five steps in the WARP development process is described in terms of the supporting object-oriented modeling approach. This is followed in Section 6 by a concrete example of the object-oriented modeling approach based on the travel agency example discussed in Section 1.

### 5 Object-Oriented Modeling for the SCW Environment

The WARP developmental process is an approach that combines human-based and agent-based modeling to gather information that can be used by agents to manage the
SCW operation. In defining the steps for the WARP development process, each of the underlying models are defined and discussed in this section.

5.1 Service Discovery and Service Capturing

Similar to the work of Benatallah [2] and Singh [17] in the area of service discovery, we adopt the ideas of elementary services and service communities. Elementary services, in the context of the WARP approach, are atomic component-based services, invocation-based services, and/or web services. In WARP, agents characterize these services and store them in an agent-accessible repository for later composition. A service community is the workflow composition of elementary services. In WARP, this service community is a virtual community because services are distributed. Accessible to the agents is a data repository that maintains the information that defines the workflow-oriented composition specifications that make up this virtual community, as described further in Section 5.2. Site Manager Agents (SMA) are responsible for the automated capturing of service characteristics in the WARP environment. The SMAs have two basic functions for gathering services, registry access and reflection. In registry access (UDDI), SMAs use the access methods provided by the registry. These access methods are relatively straight-forward, such as the find_service and get_serviceDetail methods in the UDDI specification [18].

![Fig. 4a. SMA Population Process.](image)

![Fig. 4b. WARP Service Data Model.](image)

The other function of the SMA is to gather service characteristics directly from the binary representations of the services. The SMA makes use of reflective architectures to discover the operational semantics of certain services. Reflection can be defined as a programming approach that has both a base language as well as a meta-language that describes the base language. Two such reflective approaches for component-service development are the JavaBean API and .NET. Such reflective approaches allow developers to determine the characteristics of previously deployed software. The act of determining these characteristics at run-time is called introspection. The
SMAs are able to reflectively gather operations, pre and post conditions, directly from the previously compiled services, without the requirement of having initial source code or specifications, as described in previous work [4]. An overview of the automated population procedure is illustrated in Figure 4a, in which the SMAs operate on an object-oriented data model of service information, which is gathered by the aforementioned functions. This data model (Figure 4b) consists of four entities, Component, Operation, InputMessage, and OutputMessage. The Component entity defines the components that are available for composition. A component is an independent software artifact. Multiple components may have the same name, so a unique identification (Service_ID) is associated with each. The component entity describes the component-oriented characteristics such as location and network path.

In this research, an individual component may contain multiple elementary services. Thus the Component entity is an aggregation of the Operation entity, which specifies the independent elementary services. Each service consists of a number of Input Messages and Output Messages as defined by the respective entities. In these entities, the Input/Output Messages are further defined by their data type. The Input/Output Messages contain a Value_Name field that stores the data. Since message information is heterogeneous across different types of services, to normalize the data we create this agent-adapted field using ontological approaches. Input/Output Messages of multiple services are transformed into independent ontologies, and then a consensus ontology is used to help integrate these messages. This consensus ontology is decomposed into the Value_Names. Thus agents incorporate the integrated knowledge. This approach is presented in other work [21]. Finally, the unique identification numbers are used to distinguish components, services, and input/output messages that may be named similarly.

5.2 Process Modeling

Considering the fact that, in previous steps, elementary services have been discovered, captured, and stored by the SMAs, the next step in the WARP approach is where humans intervene to model the workflow composition of these services. A service community can be defined as a repository of these compositions. In the specification of this service community, the WARP approach incorporates industry-standard modeling approaches, in particular UML. In this section, a subset of the workflow modeling semantics is described in detail.

The workflow language here follows workflow terminology used commonly by researchers. In order to set the nomenclature for further discussion, the following set of definitions are adhered to throughout this paper.

- A task is the atomic work item that is a part of a process.
- A task can be implemented with a service. (In complex cases, it may take multiple services to fulfill one task)
- An actor or resource is a person /machine that performs a task by fulfilling a service.
- A role abstracts a set of tasks into a logical grouping of activities.
A process is a customer-defined business process represented as a list of tasks.
A workflow model depicts a group of processes with interdependence.
A workflow (instance) is a process that is bound to particular resources that fulfill the process.

The WARP approach separates the semantic modeling of the workflow from the specification of services that implement the model. The task, role, process, and workflow model terms are used for workflow process specification. However, the terms, service, actor, and workflow instance, specify implementation-oriented information. This is not a new approach, but one that is necessary to ensure that the services and the workflow modeling can evolve separately.

In the WARP approach, workflow processes are specified using UML activity diagrams and class diagrams. We introduce a new approach to modeling workflow in which workflow processes are specified using multiple views, which use different UML diagrams. The advantage of this approach is that it promotes the separation of concerns in workflow specification, thereby allowing greater and more effective specificity. In addition, multiple agents can be deployed to implement the workflow process with respect to the individual concerns. Currently, this approach has been investigated using the basic workflow patterns as defined by van der Aalst [19]. These patterns include normal sequence, parallel split, synchronization, exclusive choice, and simple merge. The semantics of these patterns is similar to that provided in recent semantic web services-oriented specifications such as DAML-S, OWL-S and industry web services standards such as WSFL, BPEL4WS, and BPML [16].

There are three major concerns that are specified in WARP-based models: structural, dynamic, and nonfunctional concerns. The structural concerns deals with specification of workflow roles and how those roles are associated with specific workflow processes. From an implementation perspective, the description of underlying services must be considered in addition to their binding to workflow roles. The dynamic concerns incorporate control flow and message flow for normal workflow operation. Nonfunctional concerns consider such things as performance, atomicity, and error-handling. Nonfunctional concerns tend to be peripheral concerns important to the operation of the workflow. We adopt two groupings as specified by Kamath [15], failure atomicity and execution atomicity. In the failure atomicity grouping, models must define sequence actions that must take place when errors occur. The execution atomicity grouping includes specification of services and groups of services that are incompatible in the same workflow instance or across instances.

Using the WARP approach, the aforementioned three workflow concerns can be modeled using multiple models and views in UML, as summarized in Figure 5. The two central models are the Control Model and Role Collaboration Model. These models are described using control-based and information-based activity diagrams. One distinguishing feature of the WARP approach is the separation of control flow and message flow into two different activity diagrams. The other WARP views are the Service Representation View, the Role Association View, the Workflow Structural View, the Failure Atomicity Views, and Execution Atomicity Views. Each of these views is described in greater detail in the following sections.
5.2.1 Service Representation View
Since services are captured autonomously by the SMAs, there must be some representation such that the available services can be presented to the human workflow modeler. In the WARP approach, visual representations are presented to the human designer prior to modeling. The Service Representation View (Figure 6) allows the GWMA to represent the services available to be modeled in context of the organization housing them. This view is represented in a UML class diagram, in which the class name represents a unique identifier of the organization and class operations are used to depict the independent services available within that organization. Service composition can occur independently of organizational boundaries; however workflow modelers typically need knowledge of available organizations. In addition, this organizational name is later associated with the routing to the services.

5.2.2 Role Association View
The Role Association View (Figure 6) allows a modeler to define workflow roles based on the distributed services that the particular role will encapsulate. This view is also illustrated in a class diagram. This view builds on the Service Representation View such that additional classes are added to represent each of the roles. These classes are then associated to the service-based classes. An unnamed association assumes that all services from that organization are used or available to that role. However, by listing specific service names as the association, a subset of services from the organization can be explicitly designated.

5.2.3 Workflow Structural View
The Workflow Structural View (Figure 6) allows a modeler to specify which roles will be enacted in a particular workflow process. The workflow structural view only provides a composition of the workflow as the actual sequence of services is defined in the dynamic models discussed in Section 5.2.4. A class diagram is used that incorporates the workflow roles specified in the Role Association View. The three views provided by the structural models, as shown in Figure 6, complement each other.
5.2.4 Dynamic and Nonfunctional Models
The semantics for the Control Model and the Role Collaboration Model follow closely to related work using activity diagrams for workflow [8]. Each role is illustrated with a new UML swim lane, as illustrated in Figure 7. Each time a role executes a specific service an activity state (oval) is placed in the swim lane. The major difference that distinguishes the WARP approach is the use of the Control Model as an activity diagram that describes the sequence of actions, in addition to the Role Collaboration Model that describes the exchange of messages. In the Control Model, standard fork and join notations are used and the transitions are illustrated with solid arrows. In the Role Collaboration Model, the dotted arrow notation is used between messages. Class notations are used between the services to illustrate the individual messages. The class name represents a message as defined by the modeler. The attributes of this class are a list of the Value_Names, which act as the set or subset of postcondition messages of one service that serve as the set of precondition messages to the subsequent service.

The workflow designer may also model common nonfunctional concerns. The first concern is the assurance of failure atomicity or recovery, when some domain-related problem occurs. The Failure Atomicity Views consists of a Control Model and the Role Collaboration Model. The major difference is that default values are stipulated with the Value_Name attributes. In addition, agents can parse Failure Atomicity Views that mix control flow and message flow in one diagram. This feature was allowed because the Failure Atomicity Views tend to consist mostly of control flows after the initial exception is realized.

Agents in the WARP architecture monitor messages for the existence of these exception-driven values. The existence of these values serves as a trigger for the execution of the exception-handling-specific workflows (specified in the Failure Atomicity Views). Consequently, for each possible exception, there is a corresponding set of Failure Atomicity Views.
In addition, UML stereotypes are used to specify actions that must be taken to correct services that have been executed in the process of correcting the workflow state. Several common actions represented as stereotypes are <<abort>>, <<roll-back>>, <<roll-back and abort>>, <<re-execute>>, <<roll-back and re-execute>>, <<initiation>>. To illustrate this modeling approach in a concrete example, an incorrectly formatted customer identification scenario is shown in Figure 8. In this scenario, a customer interface web service named getUserInfo is executed. When a customer enters an invalid customer_ID the following service (verify) populates that field as not_valid. This view shows that by using the <<roll-back and re-execute>> stereotype the initial service will be reset and executed again to correct the actions.

5.3 Capturing the Process Models
In the WARP architecture, GWMAs operate on information extracted from the WARP models and views as represented in the data model in Figure 9. The main table for the process specification is the Workflow Policy table. Each record in this table defines a single process transition. A transition can be defined as the control flow between the completion of a service or group of services and the initiation of a subsequent service or group of services. These services are grouped in the Event-Grouping table. There is also a Role table that defines a role based on a group of services. The FailureAtomicity table is used to capture the nonfunctional concerns of exception-handling, atomicity, performance, and security. All tables represent the long-term storage in the run-time operation of the workflow.

5.4 Agent Self-configuration and Deployment
The final step in the WARP process is the configuration and deployment of the WARP application layer agents. Workflow Manager Agents assume workflow process responsibilities from the Workflow Policy entity in the process-oriented data repository. Through a sequence of queries, independent Role Manager Agents can be assigned roles as defined in the Role entity. Both WMAs and RMAs can automatically self-configure themselves by collecting information from the process repository via a sequence of general data queries. Since WMAs and RMAs are event-based,
they register for events, such as service completion events, which serve as the stimuli for their actions. The focus of this paper is on modeling; more details on the agent operation are given in [4].

![Customer Interface](image1)

![Account Manager](image2)

**Fig. 8.** Failure Atomicity View Example. **Fig. 9.** WARP Process-Oriented Data Model.

![Travel Agency Process](image3)

**Fig. 10.** WARP Control Model for the Travel Agency Process.

6 Composition in the Travel Domain: A Workflow Modeling Example

The use of the WARP development process is best demonstrated using a concrete example. Though a full travel agency process, as shown in Figure 10, can be modeled using this approach, in the consideration of space, only a subset can be presented. In this section, three of services from the travel agency domain are defined and modeled for workflow composition with respect to current web services-oriented specifications.
6.1 Concrete Service Information and Mapping to WARP Data Entities

The travel agency example considers a car rental service, a hotel reservation service, and an itinerary publishing service. In Figure 11, we show the car rental and hotel reservation services to be composed and integrated with the itinerary publishing service. The input and output messages are represented using sample SOAP messages and the service itself is specified using WSDL notations. These are basic examples for illustration purposes as real services would consist of more information. As discussed in Section 5.1, an agent-generated Value_Name is used as a key to describe the mapping between two like elements that are not named identically (such as ConfirmationID and RentalID). Also illustrated are the WARP data entities that relate to the particular input and output SOAP messages and the service-oriented WSDL notation.
In Figure 12a, there is a control model that shows that the car rental and hotel reservation services will be executed concurrently and the output of both services will be used to execute the itinerary publishing service. Also in Figure 12b, the class notations show the subset of information that will be transferred between the services. In these class representations, the value_names are used as opposed to the actual element names from service messages. Agents incorporate the knowledge to map similar fields with different cross-organizational naming. Information is extracted from both of these models to further populate the WARP data entities to later serve as a knowledge base for the WARP agents.

Fig. 12a. Control Model for the Travel Agency Domain.

Fig. 12b. Role Collaboration Model for the Travel Agency Domain.

7 The WARP Prototype

To evaluate the WARP architecture, a WARP prototype and experiment was developed. In Figure 13, the upper portion illustrates the high-level software design of the agent framework and lower portion illustrates the operational environment. For the initial experimentation, the prototype used only invocation-based services. Java Bean components were chosen for the implementation of component-based services. Using Java Beans, Java introspection was used to simulate UDDI calls; invocation over the Java’s RMI registry among multiple Windows 2000 workstations was used to simulate distributed services. Event-based communication was realized using JavaSpaces technology, which additionally had seamless integration to Java Beans. The
integrated data model is replicated on all machines using Oracle8iLite, a relational database management system that is operational in the personal computer environment. Java Bean services were registered both on Java’s RMI registry and given web accessibility using the Tomcat web server.

![Diagram of WARP Prototype Environment and Software Design](image)

**Fig. 13.** The WARP Prototype Environment and Software Design.

The agent architecture was developed in UML using strict object-oriented design practices. The WARP Representations were captured using the Rational Rose developer tool. Using the Rose Extensibility Interface (REI), agents access the representations via the model (.mdl) file created by Rational Rose. In this way, the Rational Rose tool was used as the workflow modeling and capturing tool.

Each type of WARP agent implicitly has all the functions of the Abstract Agent. Each agent can transmit events over the event server, register for events, and receive events. These communication functions are delegated to the Communicator class and the Listener classes. The Communicator class is used to send and receive events, while the Listener class is used to register for events. The Listener class is activated when a notification returns based on an event registration or subscription. The implementation of the full WARP prototype is approximately 20,000 lines of program code and scripting.

Important experimentation was performed on the prototype to determine the efficiency of three modes of operation. Since the WARP approach consists of a centralized database, the system can work in three modes of operation. The first mode is configuration of agents at system initialization for both service availability and business process schema. Another mode is the reconfiguration at the beginning of each
process. The final mode is re-configuration at the completion of each workflow step (service completion). For the three modes, the overhead in the WARP prototype was approximately 11%, 14%, and 25%, respectively. These figures demonstrate the additional overhead when compared with a completely hard-coded system. These numerical figures are promising with respect to reconfigurable systems that use technologies such as CORBA or COM+. Further details are given in [4].

8 Discussion

The first major contribution of this work is a reusable agent-based architecture and software process that supports the composition of distributed services. This is one of few projects that develop approaches using industry standard modeling notations such as UML [11]. Unlike related work in this area [1][8], we introduce the concept of multiple UML views to separate concerns in the modeling of the agent system that supports this SCW environment. By using multiple views and software agents to extract the operational data, service evolution and process evolution can occur independently. Furthermore, a workflow designer can control this change by visually modeling the workflow design using COTS object-oriented software engineering tools. This approach has been applied to a WARP prototype [5] where several successful scenarios were implemented with encouraging performance results. Consistency checking between these views is an important area of future work, which has been addressed by the authors in related research [10].

Earlier work by Casati and Benatallah [2][6] consider complex workflow-oriented interactions with visual and text-based specification approaches. Both adopt approaches to the service composition but have fairly rudimentary, non-agent-oriented interactions in the operation of their respective internal architectures. The work of Helal [13], Chen [7], and Singh [17] all concentrate on the agent-oriented interactions for service composition. However, the service composition specification languages used to program their agents are neither visual nor do they seem capable of handling the complex workflow interactions that are supported in our work or the support given in the work of Casati and Benatallah.

The WARP approach follows the state of the art closely with the layered approach to proxy agents, coordination agents, and specification language. In addition, our approach follows the state of the art of other related research (as Casati and Benatallah) by incorporating the use of both a visual language and a corresponding textual language for specification. We extend the approaches of Casati and Benatallah by considering modeling and agent support for both the complex workflow-oriented interactions among the services and the complex interactions of agents internal to our architecture. Prior related work has concentrated on either one or the other. Another innovation in the WARP approach is the workflow based software developmental process in the creation of these composite systems, which is consistent with industry-standard software engineering lifecycles. In fact, the use of industry-standard modeling languages, such as UML, makes the integration into industrial processes more realizable.
9 Conclusions and Future Work

This paper has described an object-oriented UML-based developmental process for modeling workflow-based service composition and incorporating a distributed agent architecture. Since input/output messages can be represented in heterogeneous formats, such as plain text, concrete objects, or XML, modeling approaches and agent support have a challenge to create operational patterns to deal with these formats. Another problem is support for heterogeneous methods of error-handling. The approach taken in this work is to create new models for each error-handling case. We believe these cases could be more efficiently integrated into the standard operational models. Future research would investigate extending this approach with advanced workflow patterns and interactions. Further research is also to investigate agent-based negotiation and performance modeling of the Quality of Service scenarios.

References

5. Blake, M.B. and Gomaa, H. “Object-Oriented Modeling Approaches to Agent-Based Cross-Organizational Workflow”, Workshop on Software Engineering for Large-Scale Multi-Agent Systems in conjunction with the International Conference on Software Engineering (ICSE2003), Portland, Oregon 2003
Using the MAS-ML to Model a Multi-agent System

Viviane Torres da Silva, Ricardo Choren, and Carlos J.P. de Lucena

PUC-Rio, Computer Science Department, Software Engineering Lab
Rua Marquês de São Vicente, 225 – Gávea
22453-900, Rio de Janeiro / RJ, Brazil
{viviane,choren,lucena}@inf.puc-rio.br

Abstract. The current object-oriented development practice system analysis is documented through UML artifacts such as Class and Sequence diagrams. Since UML is a widely accepted modeling language, it also would be desirable to offer a UML support for the representation of agent-based system analysis artifacts. Although some central UML constructs are suitably matched for agent-based modeling, several improvements must be made to the UML meta-model to achieve this new goal. This paper presents MAS-ML, a UML extension for agent-based system modeling. The use of MAS-ML for modeling agent-based systems is presented with a simple illustrating application.

1 Introduction

An important issue in getting the technology into the mainstream of software development is the development of appropriate methodologies for engineering agent-oriented software. In particular, modeling languages for multi-agent systems should explore the use of agents and other abstractions as first order modeling elements.

In the current state-of-the-art, where the concepts and notations for designing agent systems are still not agreed, we believe it is best to consider possibilities within the commonly used methodologies for traditional systems as a way to reduce the risk of adopting a new technology. Thus, UML would be a good starting point since it is widely accepted as a de facto standard for object-oriented modeling. To decrease the learning curve, we believe that a new modeling language should preferably be an incremental extension of a known and trusted modeling language.

Nevertheless, in its original form UML provides insufficient support for modeling MASs. Among other things, the UML meta-model does not provide support for modeling agents, organizations and agent roles. Using stereotypes (the UML extension mechanism) is not satisfactory because stereotypes can only be used to indicate a difference in meaning or usage between two model elements with identical structures. Based on the definition presented in the UML specification, stereotypes may not be used to represent two completely different abstractions. We realize that current OO methodologies are not sufficient; however, we believe they are relevant.

This paper motivates and presents the MAS-ML (Multi-Agent System Modeling Language) modeling language for developing agent systems. MAS-ML provides a
real agent-based extension to UML. This is achieved by extending the UML meta-model based on the TAO (Taming Agents and Objects) conceptual framework (meta-model) [9], so that the meta-model could incorporate agent-specific concerns. TAO provides an ontology that makes it possible to understand the abstractions, and their relationships, used to support the development of large-scale MASs. The ontology associates well-accepted abstractions, such as objects and classes, with other abstractions, such as agents, roles and organizations. TAO was developed based on methodologies and modeling languages [7, 8, 12], implementation frameworks (Jade, Zeus, Retsina), applications described in the literature and organization, role and MAS theories. TAO was chosen because it describes the essential concepts of MASs and their relationships.

MAS-ML uses UML as a general modeling platform. MAS-ML’s purpose is to offer extensions to the UML meta-model in order to incorporate new agent modeling capabilities, thus being a concrete proposal for improving current versions of the UML. This paper provides an example of MAS-ML to illustrate the impacts of the proposed extensions on an agent-based system modeling application. The main difference between our approach and others described in the literature is the clear definition and representation of the elements that compose MASs and their behavior.

The paper is structured as follows. Sections 2 and 3 briefly present TAO and MAS-ML, respectively. Section 4 describes a modeling approach used to perform the system design using the MAS-ML. Section 5 defines the virtual marketplace example and presents the models generated using the MAS-ML. Section 6 provides some comparisons between our approach and related work. Finally, Section 7 offers some conclusions and highlights some ongoing work.

2 The TAO Meta-model

The TAO meta-model provides an ontology to capture the foundations for agent- and object-based software engineering. Hereafter, we introduce the static and dynamic aspects of the meta-model, which are related to MAS-ML features in this paper; see [10, 11] for more details.

2.1 TAO Static Aspects

The static aspect of the TAO meta-model captures a rich variety of concepts involved in the structure of a multi-agent system; namely, object, agent, organization, object role, agent role, environment and event.

Object. An object is a passive element in the domain whose instances have state and behavior. An object may evolve from state to state. However, it has no control over its behavior, meaning that it does whatever any other element asks it to do and only when it is asked.
Agent. An agent is an autonomous, adaptive and interactive element in the domain whose instances are expressed through mental components such as beliefs, goals, plans and actions. An agent acts as a processor for plans of actions that are executed, using its beliefs, to accomplish a goal. A goal is an objective the system should meet. And a belief is some knowledge about the system.

Organization. An organization is an element in the domain whose instances group agents, objects and other organizations (sub-organizations). An organization has goals, beliefs (as agents) and axioms. An axiom characterizes global constraints that agents must obey. An organization also is responsible for defining the roles that will be played by agents, sub-organizations and objects. At least one organization must inhabit the environment. We call this organization main-organization. The main-organization is the unique organization that does not play any role since it is not defined as a member of any other organization.

Object Role. An object role guides and restricts the behavior of an object through the description of a set of features that are viewed by other elements. An object role may restrict access to the state and behavior of an object instance. But it may also add information, behavior and relationships to the object instance that plays the role.

Agent Role. An agent role guides and restricts the behavior of an agent through the description of a set of goals, beliefs and actions. An agent role defines duties, rights and protocols that restrict an agent instance. A duty defines an action that must be executed by an agent instance; a right defines an action that may be executed by an agent instance; and a protocol defines an interaction between an agent role and other elements.

Environment. An environment is an element in the domain whose instances are the habitat for agents, objects and organizations. An environment can be passive as an object, or active as an agent.

The TAO meta-model defines relationships in which the above concepts may be involved. These are:

Inhabit. Specifies that the element that inhabits is created and destroyed in the habitat and may leave and enter habitats, respecting the habitat permissions. Inhabit is applied to environments and agents, environments and objects and environments and organizations.

Ownership. Specifies that an element is defined in and must obey a set of constraints defined by another element. The member element does not exist outside of the scope of its owner. Ownership is applied to roles (members) and organizations (owners).

Play. Specifies that an element playing a role assumes its properties and relationships. The behavior of the element is guided by and restricted to the scope of the role. Every agent or sub-organization plays at least one role in an organization. Objects also can play roles.
Specialization (Inheritance). Defines that a sub-element that specializes a super-element inherits all the state and behavior associated with the super-element. A sub-element also may add and redefine the properties and behavior associated with the super-element. Specialization may be used between objects, agents, organizations, object roles and agent roles.

Control. Defines that a controlled element must do anything that a controller element requests. An agent role can control another agent role or an object role. Object roles only can control another object role.

Dependency. Defines that an element (client) may be defined to be dependent upon another one (supplier) to do its job. In other words, the client cannot completely do its job unless it asks the supplier. An agent role can depend on another agent role and an object role can depend on another object role.

Association. Defines how one element interacts with another, indicating that these elements know each other. Associations may be used between (i) roles (object or agent roles), (ii) environments, (iii) objects, (iv) agents and objects and (iv) organization and objects.

Aggregation. Defines that an element is part of an aggregator. The aggregator may use the functionality available in its parts. This relationship may be applied between object roles, agent roles, objects, agents and organizations.

2.2 TAO Dynamic Aspects

The dynamic aspects of the TAO meta-model describe the relationships between its static elements. They can be classified as primitive (elementary) dynamic processes and high-level dynamic processes.

Primitive processes describe the most basic domain-independent interactions that exist between elements. The processes of creating and destroying the elements of MAS are characterized as primitive processes. These processes define the actors, preconditions, execution steps and post conditions involved. They encompass processes for object, object role, agent, agent role, organization and environment creation and destruction [11].

High-level dynamic processes are more complex domain-independent behavior that are described based on primitive and other high-level dynamic processes. They derive from the characteristics of the relationships between entities that are associated with domain-independent behavior: ownership, play and inhabit relationships. They encompass processes for an agent entering or leaving an organization, an organization entering or leaving another organization, and an agent or an organization entering or leaving an environment.
3 The MAS-ML

MAS-ML is a modeling language that makes additive extensions to UML to provide support for multi-agent systems modeling. By augmenting the UML meta-model, new modeling capabilities were incorporated into UML, contributing to its evolution. The UML meta-model extensions proposed by the MAS-ML are based on the concepts defined in the TAO.

3.1 MAS-ML Static Aspects

The definitions of the object and event concepts of the TAO meta-model are similar to those of the Class and Event UML meta-classes; thus their notation is preserved in MAS-ML. However, it is necessary to create new meta-class definitions for the other TAO concepts. MAS-ML has included the AgentClass, OrganizationClass, EnvironmentClass, ObjectRoleClass and AgentRoleClass meta-classes to the UML metamodel [10]. In addition, some stereotypes and meta-classes were created extending the UML meta-model to represent goals, beliefs, plans, actions, axioms and messages sent and received by agents.

Each meta-class that defines an entity has a corresponding notation in MAS-ML. Similar to the diagram element that defines a class in UML, the new diagram elements are composed of three compartments (UML meta-model terminology) separated by horizontal lines. The top compartment holds the class name that must be unique in its enclosing namespace. The middle list compartment holds the list of structural features and the bottom list compartment holds a list of behavioral features. Either or both of the middle and bottom compartments may be suppressed when necessary. For example, the diagram element that models an AgentClass defines the name of the agent in the first compartment, the goals and beliefs of the agent in the middle compartment and its plans and actions in the bottom compartment. Furthermore, new diagram elements have been created and associated with new relationships defined in TAO that do not exist in UML. These relationships are Inhabit, Ownership, Play and Control.

MAS-ML extends the Class diagram to include modeling information about agents. This diagram shows the association, aggregation and the specialization relationships of TAO. Moreover, MAS-ML defines two other diagrams: Organization and Role diagrams.

The Organization diagram models the system organizations identifying their habitats, the roles that they define and the elements – objects, agents and sub-organizations – that play those roles. This diagram shows the ownership, play and inhabits relationships of TAO. The Role diagram is responsible for clarifying the relationships between the agent roles and object roles. This diagram shows the control, dependency, association, aggregation and specialization relationships of TAO.
3.2 MAS-ML Dynamic Aspects

MAS-ML proposes an extension to the UML Sequence diagram to model the dynamic aspects based on the TAO meta-model. MAS-ML extended Sequence diagram has three new elements to represent the agent, the organization and the environment concepts. Furthermore, it proposes new ways to define pathnames that identify element instances. Sequence diagrams illustrate (i) agents and organizations committing to roles and changing their roles, (ii) agents and organizations sending and receiving messages, (iii) agents and organizations executing actions, (iv) elements calling methods of an object, (v) objects executing methods and (vi) the creation and destruction of elements.

To model the dynamic processes presented above, MAS-ML extended Sequence diagram defines new stereotypes and extends the definition of the existing UML <<create>> and <<destroy>> stereotypes. The new stereotypes are: <<role_commitment>>, <<role_cancel>>, <<role_change>>, <<role_activate>> and <<role_deactivate>> [11].

Create. This stereotype was specialized to represent the creation of agents, organizations and environments; the association of a role instance to agent and organization instances; and the creation of an object and the association of a role to the object. For instance, suppose someone wants to represent the creation of the agent instance Bob. The create stereotype associated with a message from the creator to the agent instance Bob can be used to represent the creation of agent Bob and its association to a role, e.g. the Buyer role.

Destroy. This stereotype was specialized to represent the destruction of agents, organizations and environments, and the destruction of all role instances associated with agents, organizations and objects. Suppose someone wants to represent the destruction of the agent Bob. The destroy stereotype associated with a message from the destructor to the agent instance Bob can be used to represent the destruction of Bob and the destruction of all roles that Bob is playing.

Role_commitment. This stereotype was created to represent an agent, organization or object committing to a role. It can represent an agent or an organization entering an organization to play a role but cannot represent an agent or an organization entering a new environment. Suppose that Bob wants to play the role Seller; i.e., Bob wants to commit with the role Seller while playing the role Buyer. The stereotype role_commitment associated with a message from Bob playing the role Buyer to Bob playing the role Seller can be used to represent Bob committing to the new role Seller. After the commitment, Bob is playing both the roles Buyer and Seller.

Role_cancel. This stereotype represents that a role is being canceled, i.e. an element stops playing the role. It also is used to illustrate when an agent or an organization leaves another organization. Suppose that Bob wants to cancel the role Buyer that it is playing. Bob playing the role Seller can send a message stereotyped with role_cancel
to itself playing the role Buyer. After canceling the role Buyer, Bob continues to play the role Seller.

*Role_deactivate*. This stereotype changes the state of a role that an agent or organization is playing from active to inactive. Suppose Bob does not want to cancel the role Buyer but wants to deactivate it. Bob playing any role can send a message stereotyped with role_deactivate to itself playing the role Buyer. Bob stops playing the role Buyer but the role instance is not canceled.

*Role_activate*. This stereotype changes the state of a role from inactive to active. Suppose Bob wants to activate the role Buyer that is inactive. Bob playing any other role can send a message stereotyped with role_activate to itself in order to activate the role Buyer.

*Role_change*. This stereotype represents an agent or an organization changing its role. An object does not change from one role to another because it does not have the autonomy to choose its roles. Suppose Bob wants to change one of its roles; i.e., Bob wants to cancel the role Seller and commit with the role Manager. While playing the role Seller, Bob can send a message stereotyped with the role_change to itself playing the role Manager. The role Seller is canceled and the role instance Manager is created.

4 The Modeling Approach

This section presents a modeling approach to guide the understanding of the use of MAS-ML for modeling a realistic example presented in Section 5. The modeling approach describes the elements that must be defined in order to create each MAS-ML diagram. The static aspects of an application are modeled using the three static diagrams proposed by MAS-ML – Organization, Role and Class diagram. The dynamic aspects of an application are modeled using the dynamic Sequence diagram.

4.1 Organization and Role Diagrams

An Organization diagram models an environment, an organization, its sub-organizations, the roles defined in the organization and the elements that play those roles. Therefore, an Organization diagram is completely modeled when (i) the environment and the main-organization have been defined, (ii) the object roles and the agent roles defined in the main-organization have been identified, (iii) its sub-organizations have been defined and (iv) the agents and objects have been identified.

However, the Organization diagram does not describe the relationships between the roles. The Role diagram complements the Organization diagram by modeling the relationships between object roles and agent roles. A Role diagram is completely modeled while all the object roles and agent roles played by objects, agents and sub-organizations have been defined.
Environment and Main-organization Identification. It is important to analyze the environment characteristics while defining it. If the environment is a passive element it should be modeled as an object and its attributes and methods should be defined. However, if the environment is an active element it should be modeled as an agent and its goals, plans, actions and beliefs should be specified.

The definition of a main-organization comprises the specification of its goals, plans, actions, beliefs and axioms. It also is necessary to define the roles defined in the organization according to the definition of the main-organization axioms.

Roles Identification. An object role is defined by its attributes and methods. An object role can add attributes to the object that play the role and can also restrict the access other attributes defined by the object. The object role can add some methods and can also restrict the access to other methods defined by the object. An agent role is defined by its goals, duties, rights and protocols. The goals should be analyzed before other properties since they influence these properties. Protocols specify messages sent and received when two associated roles are interacting. During the protocols analysis, the duties and rights that agents and sub-organizations must obey while playing the role may also be detailed. Protocols, duties and rights must obey the axioms specified in the organization.

Sub-organizations Identification. The agent roles previously defined can be so complex that they should be played by sub-organizations. If there are sub-organizations, they must be defined; i.e., the goals, plans, actions, beliefs and axioms must be specified. Its roles must also be identified for each sub-organization. Each sub-organization will have its own Organization diagram.

4.2 Class Diagram

A Class diagram models the relationships between objects, agents, organizations and environments. A Class diagram is completely modeled when (i) the environment has been identified, (ii) the main-organization and all its sub-organization have been defined and (iii) the agents and objects have been identified.

Agents and Objects Identification. An object is defined through the specification of its attributes and methods. The definition of an agent involves the specification of its goals, plans, actions and beliefs. The roles associated with an agent influence its definition. The goals of the roles that an agent plays are related to the goals of the agent. The actions and plans of the agent are directly associated with the duties, rights and protocols defined in the roles. For instance, some actions must be associated with the ability of sending and receiving the messages described in protocols.

4.3 Sequence Diagram

A Sequence diagram models the interactions between (i) agents playing roles, (ii) organizations playing roles, (iii) environments and (iv) objects while either playing
roles or not. Therefore, the Sequence diagram depends on the identification of all elements that are defined in the MAS application.

5 A Modeling Example

A virtual marketplace was modeled to illustrate the use of the MAS-ML in a practical application. An electronic commerce example was chosen since it is referred to in the literature [4, 6] as an appropriate example of a multi-agent system.

The virtual marketplace is composed of a main-market where users are able to negotiate any type of item. In addition, the main-market defines two market types that negotiate items with particular characteristics. The market of special goods negotiates expensive high-quality items and the market of used goods negotiates low-priced, low-quality items. Users can buy items in the main-market, in markets of special goods or in markets of used goods. The users can also sell their items in the markets of used goods. In the main-market and in markets of special goods users buy the items available in the market.

In the main-market and in markets of special goods, the user looks for a seller and sends it a description of the desired item. The seller, created by the market to negotiate with the buyer, is responsible for verifying if there is an item with the same characteristics in the environment (the virtual marketplace). The environment stores all the items to be sold in these markets. If the item is found, the seller negotiates with the buyer.

In markets of used goods, the sellers and the buyers are users. The users that want to sell items announce them. And the users that want to buy items look for announcements in the market. If the buyer finds the desired item, it starts a negotiation with the seller.

The marketplace example will be expressed following the modeling approach defined in Section 4. The identification of the MAS elements presented in the example will be defined in order to create static and dynamic diagrams.

5.1 Static Diagrams

The organization model presented in Figure 1 illustrates the main-organization. In order to create this diagram the environment and the main-organization were defined together with the roles, the sub-organizations and the elements that play the roles.

Organizations are shown as a lozenge shape (a symbol with horizontal sizes and convex top and bottom). The diagram presents the main-organization General Store and two sub-organizations, Imported Bookstore and Second-hand Bookstore. Agents are represented as a rounded rectangle and the agent roles as a solid rectangle with a curve on its bottom. Two agent types were modeled in this system, user agent and store agent. The diagram also illustrates the seller and buyer roles defined by the main-organization and played by store agents and user agents, respectively.
The object roles are represented as a solid rectangle with an angle in its left corner. The diagram shows the object *item* and the *desire* and *offer* roles that it can play. The *ownership* relationship between the organization and its roles is shown as a double line linking the owner (main-organization) to the member (each role). The *play* relationship between an element and a role is shown as a simple line linking the element that plays the role to the double line that describes the ownership relationship.

The environment *Virtual Marketplace*, modeled as a passive element, is shown as a package that brings together all the entities that inhabit it. The inhabit relationship is shown by inserting the citizen in the lower compartment defined by the diagram element associated with the environment. All diagram elements were illustrated using the simplified representation that suppresses the middle and the bottom compartments.

**Environment and Main-organization Identification.** From the problem description, it is possible to identify a main-organization inhabiting the *Virtual Marketplace* environment. The environment is a passive element modeled as an object that stores items to be negotiated as one of its *attributes*. It implements the get and set *methods* to access these items. Since the user can move from one marketplace to another, the environment may also store information about other environments.

The main-organization *General Store* will represent the system main-market. Since the *General Store* is the main-organization of the system it does not play any role. As users can buy items in the main-market, the main-organization defines the *buyer* and *seller* agent roles. These roles will be detailed below. The main-organization *goal* is the management of sellers and orders. To achieve its *goals*, the main-organization defines *plans* to (i) create sellers to negotiate with buyers, (ii) to update the environment to inform that an item is no longer available and (iii) to evaluate the profit that results from sales. To guarantee that the main-organization will receive the information about all sales, an *axiom* is defined. The main-organization *beliefs* are related to the information about buyers, sellers and sales.

**Roles Identification. Roles Played by Sub-organizations.** Together with the *buyer* and *seller* roles, the main-organization *General Store* also defines the *market of special goods* and *market of used goods* roles played by sub-organizations. Since the markets of special goods sell expensive goods, these markets check if the users that want to enter the market can afford the items they want to buy. The management of new buyers is one of the market’s *goals*. Another goal is to manage the creation of sellers when buyers enter the market. The market of special goods also defines *protocols* to guide the interactions (i) between itself and new buyers, (ii) between itself and its own buyers and (iii) between itself and its own sellers.

Users can buy and sell items in the market of used goods. The market does not control sellers since they represent users selling items that are not stored in the environment. A seller announces items in the market and buyers search for these announcements. The market of used goods does not define any restriction to the entrance of new buyers or new sellers. The markets of used goods’ *goals* are to manage...
announcements and to evaluate profits. The *protocols* that the market of used goods defines are related to the interactions (i) between itself and new buyers, (ii) between itself and its sellers and (iii) between itself and its buyers.

**Sub-organizations Identification.** Examples of organizations that may play the market of special goods role are imported bookstores and exquisite goods stores. Examples of organizations that may play a role as markets of used goods are second-hand clothing stores and second-hand bookstores. In this paper we will model the imported bookstore and second-hand bookstore. These organizations define *buyers* and *sellers* that are different from those previously defined in the main-organization, namely, *buyers* and *sellers of imported books* and *buyers* and *sellers of second-hand books*, respectively.

Since the imported bookstore plays the role called market of special goods, its goals must be compatible. Moreover, since the market of special goods role is defined in the scope of the main-organization, the imported bookstore must obey its *axioms*. To guarantee that its sellers will send it the information about the sales, the imported bookstore also defines an *axiom*.

The *goals* of the imported bookstore are to manage the inclusion of new buyers in the market, to manage the creation of sellers and to manage the orders. To achieve its
goals it defines plans (i) to negotiate the entrance of a new buyer, checking if the buyer has the needed characteristics, and to register the new buyer (according to protocols that define the interaction between itself and a buyer and between itself and a buyer of imported books), (ii) to create the seller of imported books and (iii) to evaluate the profit, receive sales information and to update the environment (according to protocols that defines the interaction between itself and a seller of imported books and according to the axiom that the organization defines). Then, it sends information to the main-organization by following the duties specified in the role. The beliefs of the imported goods store are its buyers, sellers, sales and the main-organization.

The goal of the second-hand bookstore (which plays the market of used goods role) is to manage the announcement of the items to be sold and to evaluate profits. The second-hand bookstore has to follow the axioms defined in the main-organization (to send information about the sales). To do so, it defines axioms to guarantee that its sellers will send an amount of money related to sales. The second-hand bookstore defines plans (i) to store the announcements, (ii) to send them to buyers and (iii) to receive the monies generated by the sales. Its beliefs are related to the sales and to the announcements.

Figure 2 presents an Organization diagram modeling the second-hand bookstore. For space reasons, this article will not present the Organization diagram of the sub-organization of the imported goods store.

![Organization diagram modeling the second-hand bookstore](image)

**Fig. 2.** Organization diagram modeling the second-hand bookstore

**Roles Identification. Roles Played by Agents and Objects in the Main-organization.** As we have already seen, the main-organization defines the buyer and seller roles whose goals are to buy and sell items, respectively. To achieve these goals, they negotiate items stored in the environment. The buyer and seller roles define a “simple negotiation” protocol that describes how the elements playing these roles should interact. This protocol defines that a buyer asks a seller the price of an item. After consulting the environment, the seller sends the price to the buyer and the
buyer can accept or reject the seller proposal. The choice of accepting or rejecting a given seller proposal is one of the buyer role rights. If the buyer accepts the seller proposal, the seller sends the bill to the buyer. Then, the seller sends the information about the sale to the main-organization, as specified in its axioms.

Buyers and sellers have different views of the items they negotiate. An item is a desire to buyers and an offer to sellers. The desire and offer object roles have different characteristics. Suppose the item negotiated in the marketplace is a book. A book has a set of attributes (title, author, ISBN, price) and a set of methods (getters and setters to each attribute). The desire role allows the buyer to set the title, author, and ISBN of a book, but it allows the buyer to get only the price. Since the environment stores items to sell and informs the seller about these items, the seller only needs to manipulate the price attribute. The offer role allows the seller to get the price of an item since the seller must generate a bill based on it.

Roles Identification. Roles Played by Agent and Objects in Stores of Imported Books and in Second-hand Bookstores. These organizations also define buyers and sellers. However, they are different from those previously defined in the main-organization, namely, buyers and sellers of imported books and buyers and sellers of second-hand books, respectively. All the buyers and the sellers in the system have the same goals: to buy and to sell an item. However, they define different duties, rights and protocols. In order to generate the Role diagram illustrating the relationships between the roles, the protocols that they defined are described.

The buyer defined in the main-organization defines the “simple negotiation” protocol with the seller and the “entering organization” protocol with both the market of special goods and market of used goods. The buyer of imported books specifies (i) the “registration” protocol with the markets of special goods, (ii) the “searching for seller” protocol with the market of special goods and (iii) the “simple negotiation” protocol with the seller of imported books. Finally the buyer of second-hand books defines (i) the “registration” protocol with the market of used goods, (ii) the “simple negotiation” protocol with the seller of second-hand books, (iii) the “complex negotiation” protocol with and the seller of second-hand books and (iv) the “searching for announcement” with the market of used goods.

The seller defined in the main-organization only specifies the “simple negotiation” protocol with the buyer. The seller role is extended by the seller of imported books because the seller of imported books also defines the “simple negotiation” protocol. The seller of imported books defines the “simple negotiation” protocol with the buyer of imported books and the “register sale” protocol with the market of special goods. Moreover, the seller of second-hand books extends the seller of imported books role, additionally defining the “complex negotiation” protocols with the buyer of second-hand books and the “announce” protocol with the market of used goods.

Since the items sold in imported bookstores and in second-hand bookstores are different it is necessary to define new objects to represent them. The imported book and the second-hand book have the same properties associated with a book and additional attributes to indicate the book’s country of origin and an attribute to indicate the
book’s appearance. New desire and offer object roles should be defined due to the creation of these new objects. The desire of imported books and offer of imported books roles extend the desire and offer roles, respectively, including methods to access the origin country attribute. The desire of second-hand books extends the desire role including methods to access the appearance attribute. Moreover, since the books sold in second-hand bookstores are not stored in the environment, the sellers of second-hand books should be able to access all the attributes of those books. Therefore, the offer of second-hand books object role extends the offer object role defining methods to access all the attributes of a book. Figure 3 illustrates a Role diagram that emphasizes the relationship between agent roles and object roles (Part I) and also the specialization relationship between object roles and agent roles (Part II).

![Role diagram](image)

**Fig. 3. Role diagram**

**Agents and Classes Identification.** As described before, the items being sold in the organizations are books. There are three kinds of books: simple book, imported book and second-hand book. Imported books and second-hand books have similar properties described in the simple book class. Additionally, the imported book class defines an attribute to describe the item origin country and the second-hand book class defines an attribute to indicate the item appearance.

At this point, it is also necessary to describe the agents that will play the agent roles. As already mentioned, there are two different kinds of agents. The user agent represents the users in the system. A user agent is created when a new user wants to
buy or sell an item. The user agent goals depend on the user goals. To achieve its goals, the user agent may play the roles of buyer, buyer of imported books, buyer of second-hand books and seller of second-hand books. The user agent may have plans associated with its roles and goals. The user agent may, for instance, define plans to buy an item in the main-market and to sell an item in the second-hand bookstores. Plans are related to protocols defined in the role the agent is playing and they respect the duties and rights defined in this role. The user agent beliefs store the user preferences and information related to the agent experience.

The store agent represents the system preferences and it is created whenever a user wants to buy an item. The store agent has a unique goal; that is, to sell an item. The agent can play the seller and seller of imported books roles. Independent of the role that the agent is playing, it executes the same plan associated with its unique goal.

The Class diagram illustrated in Figure 4 completes the modeling of the system static aspects. The Class diagram specifies the relationships among objects, among organizations and among environments. The environment has a self-reference relationship since agents can move from one environment instance to another. Agents are not modeled in this diagram because they do not directly interact with objects, organizations and other agents. All interaction takes place through their roles. The imported book and second-hand book objects specialize the book object. The store of goods organization groups the imported bookstore and the second-hand bookstore organizations.

5.2 Dynamic Diagrams

The dynamic aspects of MAS model the interactions between agents, organizations, environments and objects. To illustrate the use of the MAS-ML Sequence diagram, two dynamic features are modeled: (i) a user agent, playing the buyer role, entering a second-hand bookstore to negotiate with a seller of second-hand books (Figure 5); and (ii) a user agent moving from one environment to another to play a buyer of second-hand books role (Figure 6). The user agent has the ability to move from one organization to another and to move from one environment to another as specified in the problem definition.

In Figure 5 the user agent instance called Bob moves from the main-organization Wall-Market to a second-hand bookstore called Siciliano. These two organizations inhabit the same virtual marketplace called Place I. The user agent interacts with the second-hand bookstore Siciliano to check if its goals are compatible with the goals of
the roles defined in the organization. The agent requests permission to play the buyer of second-hand books role. Second-hand bookstores always allow an agent to enter and play one of its roles. The agent changes its role by canceling the buyer role and creating the buyer of second-hand book role. The act of changing roles is represented by the stereotype <<role_change>>. The agent playing the buyer of second-hand books role requests the announcement of an item. The organization sends the announcement to the buyer and the buyer negotiates the item with the seller.

![Sequence diagram](part1)

**Fig. 5.** Sequence diagram (part 1)

The shapes that represent the elements in the MAS-ML Sequence diagram are similar to the shapes that represent the elements in the static diagrams. Objects are shown as rectangles, agents are shown as rounded rectangles and organizations are shown as lozenge shapes. The environment also is shown as a rectangle because it is a passive element. Furthermore, to completely specify the elements participating in an interaction we must mention the pathnames associated with them. Different elements have different pathnames [11]. For instance, the pathname that describes an agent specifies (i) the agent instance, (ii) the role instance that the agent is playing, (iii) the organization hierarchy where the agent is playing the role, (iv) the environment that it inhabits and (v) their corresponding class names. The information about the role, organizations and environment can be suppressed and a simple pathname can be used.

Suppose the buyer has tried to negotiate with all sellers and that Siciliano is the only second-hand bookstore in Place I. The buyer alternative is to move to another environment to try to buy the item. Figure 6 demonstrates the user agent Bob moving from the environment Place I to Place II.

Before moving from one environment to another, the agent must check whether it can leave the organization where it is playing roles and can leave its current environment. Then, it requests to its current environment the address of another environment.
The agent interacts with the other environment to check if it can enter it. After the environment accepts the user agent, it requests an organization to play a role. The process of choosing a role to play in an organization already has been explained in the description of Figure 5. After choosing the role, the agent moves from the organization *Siciliano* in *Place I* to the organization *From A-Z* in *Place II*. The agent deactivates the buyer role and creates the buyer of second-hand book role.

In order to simplify the modeling example, we chose not to represent the actions or the internal methods executed by the elements in the Sequence diagram. However, the Sequence diagrams represent the messages being sent by elements and the method being called by elements.

6 Related Work

Several modeling languages and methodologies have been proposed for agent-based software engineering (e.g. [2, 7, 8, 12]). GAIA [12] is a methodology for the development of agent-based systems. Gaia is agent-centered but it lacks a detailed design process, which was intentionally absent due to a desire for generality. This means that it does not provide sufficient support for the implementation of agent systems.

The Tropos methodology [7] covers the development of agent-based systems from early requirements to detailed design. However, its detailed design is oriented very specifically towards JACK as an implementation platform while MAS-ML is more general; i.e. it does not specify an implementation platform. Compared to MAS-ML, Tropos provides an early requirements phase, which MAS-ML also does (by eliciting agents, roles and so forth). MAS-ML provides a more detailed process – particularly in the architectural design phase (with the role, class and sequence diagrams).
Some other approaches are also based on taking UML and extending or modifying it, as is done by [2, 8]. This approach is sometimes justified by the observation that agents are just a special case of active objects. AUML [1, 5, 8] is a UML extension for MAS proposed by FIPA [3]. Nevertheless, AUML does not describe, in its models, inherent MAS entities such as environment and organization. As a consequence, AUML does not allow the specification of the static relationships between organizations, roles, agents and environments. Therefore, the dynamic behavior modeled simply as interactions is not sufficient for modeling an agent-based solution.

MESSAGE [2] creates specific diagrams to represent agents and other MAS elements, adding these diagrams to the set of UML diagrams. Although there are similarities between MAS-ML and MESSAGE for specification, MESSAGE uses unaltered UML diagrams for further agent specification and this brings an enormous difficulty in conceptualizing agent applications and ultimately in building good agent systems.

Just extending or using plain UML does not provide sufficient assistance to start thinking in a different paradigm. This is the reason why MAS-ML introduces modifications to the UML meta-model, using TAO concepts. It is valuable to have specific agent concerns clearly defined in the meta-model, so that the developer has no troubles in conceptualizing what are the characteristics of the agents and what are those of the objects in an MAS.

7 Conclusions and Ongoing Work

Rich and precise representation of concrete problem domains, using an agent-oriented modeling language, is an important success factor for agent-based development. This is important to ensure that agent-oriented methodologies will indeed be able to be in the mainstream for the development of information systems.

MAS-ML is a modeling language that extends UML based on the definition of the structural and dynamic aspects presented in the TAO conceptual framework. In two previous papers, MAS-ML was presented detailing the extensions proposed to the UML meta-model and to the UML diagrams. In [10] the static / structural aspects of MAS-ML are detailed. The extensions to the UML meta-model are described, the static diagrams proposed by the MAS-ML are presented and the Class diagram proposed in the UML is extended. In [11] the dynamic aspects of MAS-ML are detailed presenting the extended sequence diagram.

This paper presented an example to demonstrate the usefulness of MAS-ML. The example followed a modeling approach to guide the design method using MAS-ML and to illustrate the relevance of the proposed extensions. The Organization diagram was used (i) to present the organization and its roles, (ii) to describe the agents and objects that play their roles and (iii) to define the environment that the elements inhabit. The Role diagram illustrates all the agent and object roles defined in the example and their relationships. The Class diagram presents the system’s objects and the relationships of the organizations. The Sequence diagram was used to illustrate two dynamic processes, one representing an agent entering an organization to negotiate and another representing an agent changing its environment. The diagrams show the
interaction between agents, organizations, environments and objects. They illustrate (i) the creation of a role instance, (ii) agent commitment to this role, (iii) the cancellation of the previous role of the agent, (iv) agents and organizations sending and receiving messages and (v) agents calling environment methods.

For space reasons the example could not explore all the potential of the proposed modeling language, it presents the essential features of the structural and dynamic aspects. Using the Organization, Role and Class diagrams proposed in the MAS-ML it is possible to model the essential static aspects. The MAS-ML Sequence diagrams allow for the expression of the complex dynamic characteristics of MASs as seen in the example. This is so because agents, organizations and roles are used as first order elements in the diagrams.

While further case studies and examples are needed to test and refine the language, the use of MAS-ML has demonstrated that it is a practical modeling language to support agent-oriented software specification. Although MAS-ML is adequate to model the static relationship between the elements of MAS and to model the interaction between the elements using sequence diagrams, it is not possible to model the internal execution of agents and organization using the current version of MAS-ML. It is not possible to represent the plans and the actions that agents and organizations are executing. In order to fill this gap the authors are working on an extension of the sequence diagram.

MAS-ML has been used by the authors to express more complex versions of the present example in our laboratory. We are in the process of supervising the use of MAS-ML in a number of different MAS applications. Reports of these experiments will be available by the time this book is published. The first practical conclusions point to the fact that a design requires considerable MAS expertise to be able to take advantage of all the language features. Tool support will be a key resource to make MAS-ML dissemination viable.

It is important to mention that there is no widely accepted programming language that considers agents as first order abstractions. Software development based on an agent-oriented paradigm depends on programming languages so that it may be possible to evaluate the traceability between the requirement analysis and the implementation code [8]. To fill this gap, one of our ongoing works aims at mapping the design produced using the MAS-ML to object-oriented code. This can be used to automatically generate code from an MAS-ML model.

As already mentioned, MAS-ML has a direct impact on two UML diagrams. Another interesting path for further work is to assess how the proposed meta-model extensions will influence other UML diagrams.

Acknowledgements

This work has been partially supported by CNPq under grant 140646/2000-0 for Viviane Torres da Silva. Viviane Torres da Silva, Ricardo Choren and Carlos J. P. de Lucena are also supported by the PR ONEX Project under grant 664213/1997-9, by the ESSMA Project under grant 552068/2002-0 and by the 1st art. of decree number 3.800, of 04.20.2001.
References


Abstract. Reconfigurability and mutability are features frequently found in agents operating in heterogeneous computing environment. At the same time, they pose major challenges to the software engineering process. In this paper we review these challenges and discuss their implications towards the agent oriented software engineering methodologies. We propose a set of extensions to the Gaia agent-oriented design and analysis methodology. These extensions allow the methodology to handle certain important classes of mutable systems. These results are presented in the context of the Bond system, a FIPA compliant agent framework, with support for reconfigurability and mutability.

1 Introduction

Reconfigurable and mutable systems are an increasingly frequent occurrences in today's computing landscape. As early as 1975, the Microsoft Basic interpreter for Altair contained self-modifying code, introduced to overcome resource limitations (only 4K of memory available for the interpreter). For example, a web browser is custom application, consisting of a basic framework with multiple API's for configuration and extension, and a large number of extensions (plug-ins, codecs, drivers, applets, controls, themes and so on). These extensions are usually developed by third parties, are installed and uninstalled dynamically during the lifetime of the application, and frequently changing the behavior of the application in a radical way.

Some of the changes in the functionality of the application are desired or at least approved by the user: an example of such an extension is the ability to view new media formats. Frequently, some of the effects are undesirable from the point of view of the user: some third party extensions contain spyware, pieces of code which report usage statistics and other information about the user. Occasionally, viruses and worms can use the very same extension API's. Although software upgrades are usually considered beneficial, the externally initiated upgrade to the version 2.0 of the software in the Tivo digital video recorders have actually removed functionality previously existent on the system, an action considered malicious by users, but beneficiary by the company.
While web browsers are the quintessential user driven applications, reconfigurability and mutability are even more important for autonomous agents. Many popular agent frameworks can be used to implement reconfigurable agents, and a number of recent work explicitly focuses on reconfigurability. Varela and Agha [26] proposed the SALSA language based on the actor programming paradigm. The SALSA language is compiled to Java and targets dynamically reconfigurable Internet and mobile computing applications. The SmartApps project proposed by Rauchwerger [22] takes an approach of “measure, compare, and adapt if beneficial” for scientific applications. Restructuring occurs during various stages from the selection of the algorithm to compiler parameter tuning. The Bond agent system [5,6] was one of the first Java based agent systems with support for strong mutability. A series of primitive operations performed on a multi-plane state machine supports reconfigurability of Bond agents. This mutation technique is called agent surgery [3].

The terms of mutability and reconfigurability are related but not identical. Generally, the term mutable is applied to cases when the behavior of the application is changed in a radical way, through the manipulation of “code”, while reconfiguration is a change of behaviour through mainly “data”. In practice the distinction between code and data is frequently blurred and modifications on the agent behaviour can fill the full scale between slight adaptations through the tweaking of configuration parameters to radical mutations.

Reconfigurability presents both an opportunity for developing more powerful software systems but can also be a Pandora’s box. One way to study the effects of reconfiguration is to identify and enforce invariants which are maintained during an operation. One such invariant is that a successful run in the original agent should also be successful in the modified agent. Other invariants can cover resource management, error handling, and security aspects.

Reconfigurable and mutable agents have a special appeal for highly heterogeneous systems where there is a strong interdependency between mobility and reconfigurability. For instance, the resources available on a laptop and on a cellphone differ so widely that agents cannot migrate from one to the other without being significantly reconfigured. The solution is to migrate only part of an agent to the new location and replace some of its components with ones compatible with the new environment (this leads to the concept of plasticity of user interfaces as discussed in [25]).

In the remainder of this article we will introduce a typical agent framework with support for reconfigurability and mutation, the Bond agent system (Section 2). Next we overview the challenges posed by reconfigurability for the agent oriented software engineering process and propose a set of extensions to one of the popular agent design methodologies (Section 3). We conclude in Section 4.

2 Implementing Mutability in Software Agents

2.1 Methods of Implementing Mutability in Agents

Mutable software can be implemented using a large number of techniques. There are two important classification criteria in this respect: the granularity of the mutation and the mutation technique deployed by the agent system.
The granularity of the mutation refers to the size of the smallest component being changed. *Source code level mutation* refers to performing changes in the source code of the applications. This is a well known technique for interpreted languages such as Lisp or Prolog. Recently, many popular applications targeting the Java platform, such as the aspect-oriented programming tool AspectJ [16] or code instrumentation systems such as JFluid [14] are working in this way. *Machine code level mutation* involves performing transformations in the executable code of fully compiled languages. The mutation can happen either on the executable file as residing on the filesystem or in the runtime executable format in the operational memory. *Object level code mutation* includes transformations at the level of compiled but not linked libraries. Code obfuscators [10] or orthogonal persistency systems are working this way. A special case of these applications are adaptive libraries which can decide during runtime the algorithm to deploy such as Standard Template Adaptive Parallel Library [1]. The most popular way to perform mutation on applications is at the *component level*. Components have the advantage that they are sufficiently large and their behavior is well understood and can be isolated in specifications. Also, the interaction between components happens through relatively well documented interfaces. Developers of components are usually encouraged to implement and use components as black boxes. Thus the accidental side effects, which are the most difficult aspect of source code level mutation are significantly reduced at the component level.

Our last classification criteria is based on the techniques used to perform the mutation or adaptation operation. The typical weak mutability applications are implemented through *extension API* through which external components (plugins) can be plugged to the unchanged main application. The advantage of this approach is that the main application can still retain the control through the well controlled application interface. The plugins can run in a controlled environment (such as the sandbox model of the Java applets). Another advantage is that mutation though an extension API are reversible, which is not true in general for other techniques.

Runtime *composition API’s* allow strong mutation by changing the structure of the application, through the way in which its components are assembled. In these techniques the components are considered closed primitives. We need to identify two important concepts in this approach: the nature of the components and their assembly model. In order to be suitable for mutation techniques, a component model need to allow the *runtime assembly* of components. For example, the C++ class model can be seen as a component approach but it does not allow for runtime assembly. However, C++ based component models such as Microsoft DCOM, KDE KParts or Gnome’s Bonobo does allow runtime assembly. The Java class model on the other hand, allow runtime modification, provided that custom classloaders and special access methods are used. For the Bond agent framework, the primitive components are the strategy objects while the assembly model follows the multiplane state machine model of agenthood.

In absence of a explicit API, mutability can be implemented through *reverse engineering*. Most of the time, this approach is taken for non-cooperative agents,
for externally initiated mutations. This is the approach taken by viruses, but also by many legitimate applications, such as code instrumentation applications (for the purpose of debugging or performance profiling), or some persistency applications. One can argue, that the reflection capabilities offered by many modern languages, such as Java or Python are in fact low-level API's towards reverse engineering the code (of course, useable only if code obfuscators were not used).

2.2 The Bond Agent System

The Bond agent system (currently at version 3) is a FIPA compliant agent development environment. As the high level architecture of the Bond system (Figure 1) shows, the Bond system integrates a number of high quality open source tools. The communication framework and the strategy model is built on top of Java Agent Development Environment (JADE) framework [29]. The knowledge-base model is using the the Protégé-2000 ontology editor. The scripting support is based on the Jython [30] implementation of the Python language. Reasoning is implemented using the Jess expert system shell. Besides integrating these tools into an easy to use, consistent framework, the Bond system provides an additional set of functionalities and serve as an experimental platform. A screenshot of a running Bond agent is shown in Figure 2.

![Figure 1. The design of the Bond system](image)

In this article we will concentrate on those features of the Bond system which provide explicit support for reconfigurability and mutability: the multiplane state machine model and the Blueprint agent description language.

2.3 Multiplane State Machine Framework

Most agent frameworks are based on the notion of discrete actions generated the agent framework. The components generating the actions are usually called behaviors or strategies, and they are assembled into a structure usually being a variation of the active object design pattern (reactor [23] or proactor [21]).
The activation model of the agent framework is an important component in the behavior of an agent, and it is usually a trade-off between expressivity and enforced order.

While agent frameworks such as Jade allow an essentially arbitrary composition of the behaviors, others, such as SmartAgent [12] or the Bond framework enforce a simpler framework, which is easier to manage and model. The data structure enforced by the Bond agent model for behavior composition is the *multi-plane state machine* (Figure 3). The planes of the state machine express the concept of parallelism, the simultaneously active behaviors are described by a state vector.

### 2.4 The Blueprint Agent Description Language. Mutability Support

Bond agents are assembled using the Python based Blueprint agent description language. Although strategies can be written directly in Blueprint, the main purpose of the language is the assembly of the agents from components, initialization of the knowledgebase and description of the mutation operations. The multi-plane state machine of the individual agent is assembled from strategies pre-existent in the strategy database based on a description in Blueprint.
A Blueprint specification is a set of declarative statements that describe the internal structure of the agent, i.e. its knowledgebase, set of planes, set of states within each plane and the transitions between these states.

Besides describing the initial multi-plane state machine of the agent, it can also be used to specify surgical operations on the agent by adding or removing states/transitions from the state machine by modifying the blueprint description.

There are seven primitive mutation actions allowed by blueprint language:

- Adding a state \((\text{addState})\)
- Removing a state which doesn’t have any incoming or outgoing transitions. \((\text{removeState})\)
- Adding a transition between two states. \((\text{addTransition})\)
- Removing a transition \((\text{removeTransition})\)
- Adding a plane \((\text{addPlane})\)
- Removing an empty plane. \((\text{removePlane})\)
- Replacing the strategy of an inactive state. \((\text{switchStrategy})\)

The first six operations are a complete set of operations on the multiplane state machine. In [3] we have proved invariants regarding all these operations. The rationale behind the seventh operation is, that although it can be decomposed into a sequence of the first six, this decomposition is usually very complex (removing the transitions, removing the state, re-adding the state with a different strategy, re-adding the transitions). This sequence of operations, however, breaks the conditions of most invariants which can be proven on the agent. These invariants can be more easily proven to hold in the case of a “switch” operation.

We called the type of mutability implemented this way agent surgery. While this name is intended to be taken with tongue in cheek, it is a surprisingly good description of what is actually happening. It is a change, enacted on a living agent (Bond agents do not need to be stopped for this operation), it is performed by a knowledgeable entity (as opposed to random mutation) and it consists of removing unnecessary components or adding new components / prostesis / enhancements.
We now proceed to present an example of specifying an agent through a Blueprint description, and follow up to reconfigure it using a surgical Blueprint script. The agent we are using in our sample is a simplified version of a computation steering agent, a part of a grid application framework. The agent contains an application scheduler, a monitoring system and a user interface. The Blueprint description for this agent is presented below. The structure of the multi plane state machine constructed based upon this specification is shown in Figure 3.

```
includeKnowledgeBase("Grid")

# the application scheduler
createPlane("ApplicationScheduler")
strategy = bond.strategydb.GreedyApplicationScheduler(agent)
addFirstState(strategy, 'scheduler')
strategy = bond.strategydb.ErrorHandler(agent)
addState(s,'errorhandler')
addTransition('scheduler', 'errorhandler', FAILURE)

# the monitoring plane
createPlane("Monitoring")
strategy = bond.strategydb.Monitoring(agent)
addFirstState(strategy, 'monitoring')
addState(WaitAndTransitionStrategy(agent, 1000, SUCCESS), "wait")
addTransition('monitoring', 'wait', SUCCESS)
addTransition('wait', 'monitoring', SUCCESS)

# the Java Swing based user interface
createPlane("SwingUserInterface")
s = bond.strategydb.SwingUi(agent)
addFirstState(s,'swingUi')
addState( DummyStrategy(agent), 'dummy')
addTransition('swingUi', 'dummy', HIDE_UI)
addTransition('dummy', 'swingUi', SHOW_UI)
```

The agent presented here has a Java Swing based user interface. This Java user interface library is known to be a large consumer of resources, and it is currently not available for embedded Java implementations. This would prevent the agent to migrate to portable devices. One solution is to replace the user interface with an interface based on the AWT library - which is relying on the native windowing toolkits of the platform and is available on embedded computers\(^1\). The change from a Swing user interface to an AWT one can be performed with the following *surgical blueprint script*.

\(^1\) But AWT is a considerable less advanced user interface than Swing, and it is almost never used on desktop platforms.
setPlane("SwingUserInterface")
if currentPlane()=='swingUi':
    triggerTransition(HIDE_UI)
# removing the old visual strategy
removeTransition('swingUi', 'dummy', HIDE_UI)
removeTransition('dummy', 'swingUi', SHOW_UI)
removeState('swingUi')
# adding the new, AWT based user interface
s = bond.strategydb.AWTExecutor(agent)
addState(s, 'awtUi')
addTransition('awtUi', 'dummy', HIDE_UI)
addTransition('dummy', 'awtUi', SHOW_UI)
triggerTransition(SHOW_UI)

An alternative, simpler way of transforming the agent is:

if currentPlane()=='swingUi':
    triggerTransition(HIDE_UI)
    s = bond.strategydb.AWTExecutor(agent)
    changeStrategy(plane='IntentionExecutor', state='UIStrategy', s)
    triggerTransition(SHOW_UI)

The Swing execution engine is seamlessly replaced by AWT execution engine. Such surgical operations will frequently be required to make an agent conform to a volatile environment in which it has to operate.

3 The Software Engineering Perspective of Adaptive Agents

As agent systems migrate from research laboratories to the world of commercial software development and enterprise computing, new design and analysis techniques must be developed. It becomes increasingly clear that the established software methodologies, such as object-oriented analysis and design, are inadequate or insufficient for the analysis and design of agent systems. Software engineering methodologies specifically designed for agent systems become necessary.

Some of these new developments, such as [17,27], are building upon the existing object oriented methodologies and techniques and design patterns. There are a number of efforts underway to extend the UML language and the associated software methodology for agent oriented programming ([19,9,2]) or for modelling the knowledge-base of the agents [11,13]. Many methodologies are drawing inspiration from the Belief-Desire-Intention model [17,20]. Other approaches are building on techniques for knowledge engineering [7] or on formal methods and languages, e.g., the extensions for the Z language [18]. The Tropos methodology [8] is adapting ideas from techniques developed for business process modelling and reengineering (the i* notation [28]), at the same time retaining the mentalistic notions of belief-desire-intention and related models. Some agent systems
have developed their own analysis and design approaches, targeted to the particularities of the agent system such as Cassiopeia.

The introduction of mutable agents creates new problems for the agent analysis and design methodologies. The analysis step needs to take into consideration the possibility of the agent being significantly modified during its lifetime. The design step needs to offer information about which agents should be mutated, at what moment of their lifecycle, and what kind of mutation should be performed. Generally, the methodological discipline is more important for the case of mutable agents.

We now discuss the effect of mutable agents on one of the popular agent design methodologies, the Gaia approach [27]. The Gaia methodology, with its roots in object oriented approaches such as FUSION, is a good fit for FIPA compliant agent systems such as Bond, as long as they do not mutate. In fact, the authors of [27] explicitly spell out among the applicability requirements that (a) the organizational structure of the system is static and (b) the abilities of the agents and the services they provide are static, do not change during runtime. Several extensions proposed to the Gaia methodology extend the scope of the methodology. The ROADMAP methodology [15] extends Gaia with formal models of knowledge, role hierarchies and representation of social structures. It also extends the permission attributes to allow roles to change the definition or attributes of other roles, although it does not cover the issue of how the modified agents are represented.

Our goal is to investigate the feasibility of the removal of these constraints and the changes in the methodology implied by this removal.

We emphasize that no methodology can handle randomly mutating agents. Fortunately, the most frequently encountered operations can be classified in a set of well understood classes:

- Adding new functionality to the agent (understood as the ability to function in new roles).
- Removing functionality from an agent (understood as the inability to function in a set of roles in the future).
- Adapting the agent to new requirements or a different set of available resources.
- Splitting an agent.
- Merging agents.

While many other types of change can be envisioned (for example, transferring a functionality from an agent to a different agent), we will concentrate in this paper on the five mutation types shown above.

### 3.1 Adding New Functionality to the Agent

In terms of the Gaia methodology, adding new functionality to the agent is equivalent to saying that the agent will be able to function in new roles, while maintaining the previously existing ones. More formally, considering the reconfiguration event $e$, we can say that if the agent was able to fulfill a set of roles...
$R$ before the event, and after the event, it will be able to fulfill a set of roles $R'$ with $R \subset R'$.

The corresponding structural definition in the $AM_1^{mp}$ model, employed by the Bond system, is that the extending functionality is an agent surgery operation, which transforms agent $A$ to agent $A'$ and for every run $R$ where the agent $A$ is successful, the agent $A'$ will be successful as well. In [4], we demonstrated that elementary surgical operations, such as adding states, adding transitions, and removing transitions labelled with FAILURE, maintain this property. In addition, the surgical operation of adding a plane can also maintain this property subject to a set of disjointness conditions.

We found a good mapping between the Gaia concept of roles and the structural implementation. Just as the agent might not be taking on a certain role, although it would be qualified to do it, the agent might not perform certain runs. Thus we can say that by adding new functionality to the agent by agent surgery, the agent acquires the ability to fulfill new roles.

The attributes associated with the roles in the Gaia methodology will also be maintained: the responsibilities, permissions, activities, and protocols. As these attributes are applied to the agent in a cumulative way, an important goal of the analysis process of an agent surgery operation is to determine that there is no conflict between the attributes of the agents, that is, the invariants of the agents are properly maintained.

A different question, which needs to be answered by the analysis process, is the opportunity and moment in the agent lifecycle when the new functionality needs to be added to the agent. The answer to this question is not an explicit point in the lifecycle of the agent, but a trigger, a specific set of conditions under which the mutation becomes desirable\(^2\).

For a concrete example of how the diagrams of the Gaia method can be annotated to handle reconfigurable agents, let us turn to the airline industry for an example. During a flight, the number of (human) agents on the airplane are playing a set of well defined roles: passenger, pilot, stewardess and so on. There are, however, exceptional situations, such as an emergency landing in which case some of the passengers are required to take on new roles, such as to assist the crew in opening the doors.

The agent model diagram of this situation is modelled in Figure 4. In this approach, the exceptional situation is modelled as the mutation trigger. The new state of the agent is modelled in the Gaia agent diagram as a new agent type. Mutated agent types are marked in the diagram with the letter M. The

\(^2\) Although one might argue that the agents can be designed so that they can fulfill all the possible roles needed during its lifetime. This argument ignores the cost associated with having such a multifaceted agent. To put this in a different context, not all the workers of a company need to be qualified for all the professions. Nevertheless, it is a frequent occurrence that a worker needs to be sent to additional training so that he or she can fulfill new roles. In many cases, these events can be quite accurately predicted and even planned. Similar considerations apply to agents.
mutation operation is specified using a thick arrow with the “+” label attached (indicating that the mutation retains all the previous roles of the agent\(^3\)).

One of the potential difficulties associated with this model is the potential for an agent type space explosion.

\[\text{Trigger: \quad \text{EmergencyLanding}}\]
\[\text{PassengerAgent} \quad 80..140\]
\[\text{Customer} \quad \text{DoorOpener} \quad 5..8\]

**Fig. 4.** Agent model for an airplane emergency situation

Another diagram which needs to be adapted to handle the needs of the reconfigurable agents is the acquaintance model. While the Gaia acquaintance model does not deal with the details of the interaction, mutations on the agents can frequently change the acquaintances as well. The example presented in Figure 5 also deals with a fictional situation on an airplane. The sudden symptoms of sickness on some passengers and a stewardess triggers a request from a stewardess which makes a passengers step into the role of a doctor. This creates a new interaction pattern, between the doctor and the sick stewardess and the sick passenger. These acquaintance lines would have not existed if the mutation would not have happened.

### 3.2 Removing Functionality from an Agent

There can be several reasons for removing functionality from an agent. One of them is the course-grain equivalent of garbage collecting. At some moment in the agent’s lifecycle we might find that some of the roles of the agent will never be activated. The ability to perform roles which will not be activated usually implies some kind of waste of resources. Examples are memory and disk space occupied, network bandwidth by polling for messages which will never arrive, processor time for maintaining data structures which will never be queried. It is therefore useful to periodically perform a role-based garbage collection process on the agent. While the process is related to the garbage collection process in programming languages automated programming languages such as Java, there are also some specific differences.

---

\(^3\) Strictly speaking, the “+” label is not needed on the agent model diagram, because the operation can be inferred from the role inheritance lines. It is however useful on the other diagrams where the role inheritance is not present.
The garbage collection process happens at the level of components and subsections of the knowledgebase (instead of allocated memory chunks).

Active components (code) can be also garbage collected.

The internal structure of the agents can greatly simplify the garbage collection process. For example, for agents based on a multiplane state machine model, the garbage collection process can be reduced to a reachability analysis on the state machines.

The probability that a garbage collection step will recover some resources is generally lower, then in the case of garbage collecting memory in applications. Moreover, the benefits of role-based garbage collection will tend asymptotically to zero, unless some other mutation operations in the meanwhile new components.

The role-based garbage collection can happen any time during the agent’s lifetime. However, the life cycle of an agent provides some natural points where the side effects of the garbage collection process are minimal. Such points are: after every transition (for state machine or Petri net based agents), before check-pointing, before moving (for mobile agents) and after every mutation.

Another scenario for removing functionality from agents, is to trigger specialization in groups of large agents. In this case an agent factory generates agents with the ability to perform a set of tasks. The agents are then specialized through removing their ability to perform a certain subset of tasks (Figure 6). The specialization mutation can be either performed under the control of a remote agent, or it can be performed by the agent itself, based on the initial experiences of its lifecycle. This approach is very natural for distributed solution of problems with the “divide and conquer” approach, such as the popular Contract Net protocol [24]. Another application is the emergence of communication patterns. There is solid evidence that the visual and auditory pathways of mammals are generated using a similar method in the early stages of life. The hardware industry had chosen a similar approach for the zone codes of DVD drives. The DVD drives are manufactured as generic devices, which are capable to play DVDs from any zone. During the first several uses, the DVD drives decides on a particular zone coding, and it permanently removes its own ability to play DVD’s from other zones.

Fig. 5. Acquaintance diagram for the “sickness on an airplane” scenario
3.3 Adapting an Agent to New Requirements

Another very important subclass of agent mutations are when an agent is reconfigured to adapt to a changing environment expressed with a changing set of resources. The most typical one is that the reconfiguration needed after migration to a new host computer. Another example, which does not involve migration, when an agent running on a host computer needs to adapt between the almost complete control over resources (when the user is not logged in) with only minimal resource allowances (when the user is logged in and working).

Expressed in the terms of the Gaia methodology, the agents will implement the same set of roles with a different set of attributes.

- The responsibilities and the protocols implemented by the role will remain unchanged during this operation\(^4\).
- The permissions associated with the role will be different. The Gaia methodology collects under the concept of permissions notions such as resource usage and security permissions. The analysis process is responsible to assure that the permissions required by the agent after the transformation are satisfied by its new context. For example, it has to be assured that the memory and processor power requirement are satisfied.

As in the case discussed in the previous section, the goal of the agent analysis and design is to determine the opportunity and the nature of the agent mutation. The opportunity for migration can be expressed in terms of hard and soft triggers. A hard trigger is a boolean function which tells us if an agent can not fulfill the requirements of its role in a given context. A soft trigger, on the other hand, is a cost-benefit analysis of the agent which might suggest a mutation if this leads to an increased performance of an agent. Generally, hard triggers are leading to changes with implementations with lower resource usage, while soft triggers are biased towards implementations with higher performance and associated higher resource (permission) requirements.

\(^4\) This is a relatively crude approximation which assumes that the functionality of an application is completely specified by the original specification.
The adaptation scenarios can be described in the terms of a design decision tree. In the Gaia approach (as in many other software engineering approaches) the designer moves from an initial, very high level specification, to an increasingly more specialized choice. These choices for a design tree, which will be assumed to be a separate one for every role the agent can play. For the example given in Figure 7, the choices are all valid approaches to create a user interface of the program. During the design process, a set of decisions are made to determine the permissions (in the resource usage sense) of the program. The leaf nodes typically correspond to the actual implementation of a program, but of course, not all the potential choices are actually implemented. For an agent which does not require reconfiguration, only one branch of the design tree is explored. Once the decision is made, the design tree is not used in the actual operation of the agent, although it might be kept, implicitly in the design documentation.

For a reconfigurable agent, more than one leaf node is fully instantiated. We need to emphasize that this involves the same analysis and programming steps as for any other agent. However, in this case, the design tree has a practical utility during the lifetime of the agent. Let us assume an agent which is executed with a Swing based, fully graphic user interface, and needs to be migrated to a Personal Digital Assistant. A simple check of the permissions of the role will tell us that the current implementation will not work and a reconfiguration operation is needed.

The role, therefore will be moved backwards in the design tree. At every steps releasing the assumptions made at the given point, until the assumptions on the current point in the tree do no conflict with the new context of the agent. Normally, however, we are usually at a more or less abstract specification level, which on its own, is not runnable. Therefore, we will start to move toward the leafs of the decision tree again, this time however making decisions according to the new context. Our goal is to reach a fully implemented leaf node which conforms to the current set of permissions. The required transformation will be, therefore, one which transforms from the original design choice into a new design choice.

### 3.4 Splitting and Merging Agents

Splitting and merging agents are operations which are surprisingly easily implementable in many agent systems. Moreover, very compelling application scenar-
ions can be found for them. The reason why this technique is not more frequently used in applications is because there is no accepted software engineering methodology to specify them. Also, splitting and merging is not a mechanically intuitive concept such as agent mobility.\textsuperscript{5}

The software engineering process for the splitting and merging agents involves most of the notions presented in the previous sections. We need to identify the triggers of the split and merge mutation, and the resulting new agent types need to be fit in the agent models. We propose the use of the split and merge operators as presented in Figure 8.

![Fig. 8. The agent model for splitting and merging operations](image)

4 Summary

Although need for reconfigurable and adaptable systems seems to be unanimously accepted, the use of mutable agents, in the sense presented in this paper, is more controversial. For example, adaptation can be achieved at the level of societies of agents - through the creation and termination of statically configured agents. Agents can be reconfigured with configuration files, without radical changes in the code. Further research is needed to establish the domain where mutability is more desirable than other approaches. One direction of research is the extension to other agent oriented software engineering methodologies, such as Tropos or Agent UML. Our current experiments used the Bond agent system which was explicitly designed to support mutability. One of our research directions is to determine the minimal sets of requirements against of a general purpose component system, such as Enterprise Java Beans or Microsoft .NET which would allow the use of mutation techniques.

\textsuperscript{5} Agent mobility, in practice involves: stopping an agent, serializing it, transferring data and code over the network, signalling back that the transfer was successful, destroying the agent in the original location, restarting it on the new location. It is quite obvious that this is a complex, and not entirely intuitive process which has little to do with movement in the mechanical sense. However, the power of the metaphor made people accept the notion of mobile agents easier because most of us can visualize it.
In conclusion, the software engineering of mutable agents opens many controversies and difficult theoretical and practical questions. The results presented in this paper show that one of the popular agent development methodologies (Gaia) can be extended for the design and analysis of populations of mutable agents. An unexpectedly good fit between the Gaia design methodology and mutable agents seems to exist. While new questions need to be answered and new invariants must be verified, we found no need to introduce new high level concepts. This is a good omen for the reconfigurable agent technology. The populations of changeable and mutable agents might be manageable, after all.

Acknowledgements

The research reported in this paper was partially supported by National Science Foundation grants MCB9527131, DBI0296107, ACI0296035, and EIA0296179.

References


Improving Exception Handling in Multi-agent Systems

Frédéric Souchon¹², Christophe Dony², Christelle Urtado¹, and Sylvain Vauttier¹

¹ LGI2P - Ecole des Mines d’Alès - Parc scientifique G. Besse - 30 035 Nîmes - France
² LIRMM - 161 rue Ada - 34 392 Montpellier - France
dony@lirmm.fr

Abstract. New software architectures based on multi-agents or software components allow the integration of separately developed software pieces that interact through various communication schemes. In such a context, reliability raises new important issues. This paper aims at increasing reliability in multi-agent systems (MASs) and, therefore, focuses on the study of an appropriate exception handling system (EHS). The issues specific to exception handling in MASs – preservation of the agent paradigm and support of cooperative concurrency – are presented and discussed. This paper analyses existing EHSs according to these issues and describes our proposition, the SaGe system, which integrates various solutions from existing EHSs and adapts them to the agent paradigm. SaGe is an exception handling system dedicated to MASs that addresses the stressed issues by providing means to coordinate the collective activities of agents, to embed contextualized handlers in agents and to concert exceptions. It has been implemented and integrated in the MadKit MAS. It has been experimented with a classical travel agency case study.

1 Introduction

New software architectures (such as multi-agent systems [1] or software component based architectures [2]) are based on the integration of numerous software entities being executed concurrently and, in many cases, communicating asynchronously [3,4]. We are interested in reliability [5] concerns in the context of these new architectures but this paper chooses to narrow the focus on exception handling in the context of multi-agent programming. To our opinion, exception handling capabilities are a must-have to enable the realization of reliable large scale agent systems. To be adapted to multi-agent systems (MASs), we consider that an exception handling system (EHS) has to correctly deal with two main issues:

- preservation of the agent paradigm,
- support cooperative concurrency among agents.

This paper describes an EHS dedicated to multi-agent programming which addresses both issues and extends the MadKit platform [6].
The remainder of this paper is organized as follows. Section 2 introduces the terminology and main concepts of both exception handling and the agent paradigm. Section 3 discusses the issues of exception handling in the context of MASs and discusses related systems and solutions. Section 4 describes our proposition: the SaGE system which is illustrated through a comprehensive example. Finally, Sect. 5 and 6 explain the implementation of SaGE in MADKit and present our first experiments with this implementation.

2 Basic Concepts

2.1 Exception Handling

Exceptional events, often called exceptions [7], allow to signal undesirable situations that hamper a program standard execution to continue. When such a situation occurs, a reliable software is able to react appropriately by raising and treating an exception in order to continue its execution or, at least, to interrupt it properly while preserving data integrity as much as possible. An exception handling system [8,9,7,10,11,12,13,14] provides programmers with control structures that allow him to signal exceptions, to define exceptional continuations by the means of exception handlers and to put the system back in a coherent state by coding how some detected exceptional situation has to be treated. This latter objective can be:

- achieved by the handler itself by either continuing the execution where it was interrupted after having modified the context in which the undesirable situation occurred (resumption), or aborting part of the standard execution and resuming at some reliable point (termination),
- or, delegated to another handler by either propagating the exception or signaling a new one.

Signaling an exception provokes the interruption of what is currently being executed and the search for an appropriate handler. The definition of a handler (for one or more type of exceptions) results in associating code with program units which nature can vary from a system to another (e.g. the body of a procedure [8], a class [9], etc.). After an exception has been signaled, an appropriate handler is searched for among those associated to the program unit in which the exception has occurred. If one is found, it is executed. If not, the search carries on recursively in the enclosing program unit. The set of program units the exceptions of which can be treated by a given handler is called the handler scope. Depending on the models, this scope can be determined either statically (on the basis of the lexical structure of the code) or, more often, dynamically (on the basis of the history of execution contexts). This history is being built as a program unit (the called unit or the enclosed unit) is activated by another (the calling unit or the enclosing unit). An execution context is then created and associated to the called unit; it contains information related to its execution (local data, return address, parameters, etc.). This way, the calling unit precedes
the unit being currently executed in the history of execution contexts. Systems that provide handlers with a dynamic scope rely on this history of contexts to determine the scope of a given handler at runtime. For example, the scope of a handler associated to a procedure covers the execution of the procedure itself and the execution of all the procedures it calls (and so on, recursively). This kind of mechanism is used, for example, in the C++ and Java languages.

2.2 Agent Paradigm

The main characteristic that distinguishes agents from other software components is their autonomy [15]. An agent has the ability to independently decide to realize an activity (among its capabilities) in order to fulfill individual objectives. Agents therefore execute concurrently in separate threads. However, agents are not isolated entities. They interact by exchanging messages thanks to asynchronous communication means. This way, agents are able to collaborate while preserving their autonomy.

Sending a message to another agent to request a service is a non-blocking action: the client agent (the sender of the request) does not have to wait for the response of the provider agent (the recipient of the request) and can carry on its current activity. The client agent will get the response later, in another message, sent in return by the provider agent. Conversely, receiving a message is a non pre-emptive action: the agent can decide to postpone the treatment of the message in order to achieve more urgent activities first.

Asynchronous communication provides agents with a means to manage advanced execution schemes such as to redundantly request the same service to different agents in order to ensure better performance, reliability or quality of service. For example, a client agent can calculate and consolidate a result with the different responses it gets from a pool of provider agents and then decide to stop the process when a given amount of time has been spent to collect information (to guarantee some time performance), or that a given ratio of responses has been received (to guarantee some result representativity). This can also be used to manage collective activities when agents are structured in social organizations. In Aalaadin (the social organization model used in MadKit) [16], every agent is member of a group, in which it plays one or more roles that define the different responsibilities (capabilities) of the agent. The role then acts as a common interface for a set of agents. Inside a group, a request can be sent to an individual agent or to a role. In this latter case, the request is transparently broadcasted to all the agents that play the role. The client agent can therefore get a collective response from all the members of its group that hold a given capability, while ignoring both their identity and their number.

Nonetheless, the autonomy of agents should not interfere with the concept of contract-based collaborations [12], that is a fundamental principle for reliable software engineering. Indeed, the service requested by a client agent to a provider agent may be essential to complete some activity. A provider agent is free to reject a request but when it accepts to provide another agent with a service, it must fulfill its commitment. Thus, when an agent undergoes a failure that
prevents it from executing any requested service, it must warn the concerned client agent that it is not able to provide the service it expects. The client agent will then be able to react to this situation and try some alternate way to obtain the needed service or to achieve the impacted activities.

In other words, agents need an exception handling system too, in order to take reliability issues into account in the management of their collaborations. But because of the principles of the agent paradigm, that imply the use of specific execution and communication models, exception management in MASs raises specific issues that cannot simply be addressed by the EHS of the underlying implementation languages. Section 3 presents these issues and discusses some related work.

3 Exception Handling in MaSs: Issues and Related Work

In this Sect., we study exception handling in the context of multi-agents platforms and discuss the two main issues we consider to be specific to exception handling in this context.

3.1 Preservation of the Agent Paradigm

As argued in Sect. 2.2, we consider agents from a software engineering point of view (we do not consider their cognitive capabilities) and, therefore, think that multi-agent systems, just as software coded in other paradigms, need exceptional situations to be dealt with adequate control structures.

A Need for a Specific EHS. Agents are not native concepts of the language chosen to implement the MAS. They are higher level entities that use specific communication and execution mechanisms in order to conform with the agent paradigm. The management of exceptions at the agent level therefore requires a specific EHS that is integrated and adapted to these mechanisms: exceptions should be propagated from provider agents to their client agents thanks to messages; agents should immediately react to messages signaling exceptions by searching for handlers associated with the activities that are affected by these failures. This specific EHS and the EHS of the programming language must not be mistaken. They handle different categories of exceptions (as classified by [17,18]): the latter is used to handle low-level exceptions (related to the implementation and the execution environment); the former deals with high-level exceptions, regarding the execution of the activities of agents, their interactions or their management within the MAS. But the two EHS are related: an implementation level exception that causes the failure of the execution of an activity is to be transformed into an agent-level exception that can then be handled by the EHS integrated to the execution model of the agents.

Among the existing MASs [19], very few provide a specific EHS. Most of the MASs limit their exception handling capabilities to those provided by the programming language. For example, MadKit [6] is a generic MAS coded in
Java that does not prescribe any specific execution model and, as a consequence, does not provide any specific EHS. The activities of agents are coded as methods and the exceptions raised during their execution are classically treated by the handlers associated with blocks of code of these methods. When an exception cannot be treated by these handlers, it is propagated to the top level of the call-stack of the thread in which the agent executes. The thread is then destroyed by the Java virtual machine. From the MAS point of view, this corresponds to the accidental death of the agent. The lack of an appropriate agent level EHS implies that no exception can be propagated to other agents or more simply to the MAS itself, as exceptions are not supposed to be propagated outside the execution thread\(^1\). In such a context, agent level exceptions have to be managed with ad hoc solutions such as to signal a failure to a client agent by replying to its request with a message containing special values. But this solution contradicts software engineering good practices that recommend the management of errors thanks to a dedicated means like an EHS. Indeed, separately developed agents are unlikely to have a shared interpretation of those special values and the code of activities then mixes up the treatments of normal responses – that describe the standard behavior of the agent – with the treatments of exceptional responses – that describe the corresponding exceptional behaviors used to deal with critical situations.

**Supervisor-Based EHSs Are Not Sufficient.** The specific EHSs designed for some MASs \([20,21]\) use a supervisor-based approach. Supervisors are specialized agents which role is to monitor the activities of other agents, in order to catch the exceptions they could raise. The supervisors are the entities to which handlers are attached. We advocate that this approach is more adapted to handle generic problems at the MAS level (such as the death of agents \([20]\)). Indeed, from the point of view of the monitored agents, supervisors are external entities created by the MAS. The management of the particular problems related to the inner activities of an agents requires that specific, contextual handlers (aware of the precise impacts of the failures signaled by the exceptions on the current activities of the agents) be defined and triggered. As external entities, supervisors cannot easily manage such contextual handlers, unless supervisors have some right to act intrusively on the behavior of agents or unless agents inform supervisors of specific exception handlers, and thus delegate to supervisors the management and execution of parts of their behavior. In both cases, it contradicts encapsulation, abstraction and autonomy principles.

We thus propose a different and complementary solution that is more natural to deal with contextual exception handling: handlers designed to treat the exceptions regarding activities of agents should be associated with these activities, and, by the way, encapsulated and managed by the agents themselves, as part of their behaviors. Exceptions that cannot be successfully caught and treated by the handlers of an agent are then propagated outside of the agent, whether to

\(^1\) MadKit does not use the java *ThreadGroup* notion, that would be of little help as it does not support distribution.
agents with which it collaborates as a provider for some service, or to the MAS (represented by supervisors) for more generic system-level problems.

As a conclusion, we claim that the development of a specific EHS is essential to exception handling in MASs (the underlying language exception capabilities are not sufficient for agents) and that the EHS must not solely rely on a supervisor-type architecture.

### 3.2 Cooperative Concurrency Support

A multi-agent system is made of software entities (agents) that execute concurrently. Solutions to handle exceptions in MASs can thus be derived from work on concurrent programming. This section presents two concepts proposed for concurrent programming that we adopt and adapt in our proposal for agent-based programming.

**A Need for Activity Coordination.** [22] proposes a classification that distinguishes between three kinds of concurrency in object-oriented systems and studies their impact on exception handling. This classification is provided for classical concurrent object-oriented systems. In such systems, execution threads are orthogonal to objects: objects are passive entities that are executed by external threads. Conversely, agents are active (concurrent) entities that hold their own threads and use this processing power to act as autonomous entities. Thus, in MASs, threads are not disincarnated: a thread is created for the purpose of executing a given activity of a given agent and its life cycle follows the life cycle of this activity. The classification of [22] must thus be adapted to this specificity of MASs but still applies in our case.

First, **disjoint concurrency** points out the kind of concurrency supported by systems that actually provide no way to manage concurrency. Disjoint concurrency means that each agent is managed as if it is the only active entity in the system. Exception handling is therefore local to each agent. There is no need to provide any mechanism to coordinate exception management between agents as no collective activities are considered. Second, **competitive concurrency** points out the kind of concurrency supported by systems that manage the isolation of each active entity. These systems provide mechanisms to avoid the inconsistencies caused by the concurrent use of shared resources (generally thanks to lock-based schemes). The goal of such systems is to let every agent act as if it was the only active entity in the system, but in a disciplined way. Exception handling is coherent with this concurrency policy: Exception management is still local to each agent. Finally, **cooperative concurrency** points out the kind of concurrency in systems that provide some support to the management of collaborations between active entities. In a MAS supporting cooperative concurrency, it should be possible to coordinate the individual activities of the set of agents that contribute to a collective activity. [22] claims that cooperative concurrency management requires an execution model that allows these collective activities to be explicitly represented. An EHS could then be designed that
allows handlers to be associated with a collective activity, in order to express and manage the impact of the failure of every participant agent in such a global execution context. Exception propagation schemes can then be elaborated, from the local execution context of the individual activity of an agent, to the more global execution context of the collective activity to which it participates, and recursively, to the even more global execution context of an enclosing collective activity.

To summarize, **MASs must provide support for cooperative concurrency**, as collaboration is a fundamental principle of the agent paradigm (because of the social abilities of agents). As a consequence, the execution model of MASs must provide a means to explicitly represent and control the collective activities of agents.

MASs using supervisors in their EHS [20,21] support a form of activity coordination: a supervisor can be associated to a group of collaborating agents to monitor the global activity of the group. MASs that do not provide a specific EHS must use the exception handling capabilities of their underlying programming language. Those written in Java can use an advanced notion to coordinate the threads of agents: thread groups. A thread can be attached to a thread group. When an exception is raised inside a thread, that is not caught and reaches the top-level of the call-stack of the thread, the exception is propagated to the thread group to which the thread belongs. The exception can then be caught by handlers associated to the thread group so that the impact of this exception on other threads of the group can be managed (such as to stop the threads that are dependant on some result that the faulty thread has not produced). The thread group acts as a coordinator for the activities of its belonging threads. A thread group can in turn be part of a larger thread group. This allows large, hierarchical thread structures to be build, in order to control collective activities at different nested levels. Thread groups are an interesting Java construct to coordinate concurrent thread activities but:

1. they are not used in the EHSs of the MASs we surveyed,
2. they do not provide a direct support for distribution or asynchronous messaging. Therefore, they are not sufficient, as is, to manage the coordinated activities of agents (see Point 1 in Sect. 3.1).

**Concerted Exception Support.** Once activity coordination is supported and exception handling mechanisms are integrated, a second step is to define a management policy that determines the impact of the co-occurrence of minor failures while completing a global activity. [3] and then [23] suggest the integration of a mechanism to concert exceptions. Exceptions concurrently signaled by the entities participating to a collective activity are composed together, by a resolution function, as a unique exception (called a concerted exception) that reflects the global state of the collective activity. This concerted exception is used instead of the individual exceptions raised by the participants to trigger handlers at the collective activity level. This is a general principle that can be adapted, through
different implementations of its basic concepts (concerted exception and resolution function) to the exception handling mechanism of any system supporting cooperative concurrency: the entity that is responsible for a collective activity is able to collect the exceptions that are propagated from the participant entities and then to compute a unique, pertinent global exception. Another advantage of this scheme is the versatile support for exception management policy definition it provides. A redefinition of the resolution function associated with a collective activity is enough to change its policy.

The next Sect. depicts how these concepts are adapted and used in our EHS proposal for MASs.

4 Sage: An Exception Handling System Dedicated to Multi-agent Systems

This Sect. presents our proposal: an EHS that tackles the two issues stressed in the previous section. We will use a unique comprehensive example to illustrate our model throughout the remaining of the paper. It is based on a classical travel agency case study (see Fig. 1) in which:

1. A Client agent contacts a Broker agent in order to organize a travel and get the best offer for its plane or train tickets (the transport means is chosen randomly during initialization).
2. Depending on the request made by the Client, the contacted Broker sends a request to train providers or plane providers to collect their bids.
3. Then, the Broker selects the best offer and requests both the Client and the selected Provider to establish a contract.

4.1 An Execution Model That Allows Concurrent Activity Coordination

The MAS we use for our experimentation, MadKit, is a generic MAS that does not prescribe any predefined, fixed execution model: it only provides a framework of versatile communication and management mechanisms for agents. Thus, the execution model presented in this Sect. is the one we designed for MadKit agents because we need cooperative concurrency between agents. Intra-agent concurrency is also mandatory in our system in order to preserve the responsiveness of agents to critical events such as exception signaling. Though initially designed for MadKit, the execution model presented here is generic and we propose it as a solution to manage the coordination of concurrent activities that could be transposed to any MAS.

Exception handling in MASs requires a rather sophisticated execution model. Indeed, agents must always remain responsive to messages, particularly to critical messages such as interruption calls (a client agent signals to one of its provider agents that it does not need the service it requested anymore) and exception signaling (a client agent signals to one of its provider agent that it cannot successfully treat a request because of some failure). To ensure its responsiveness,
every agent owns a thread dedicated to actively scanning its message-box in order to be able to trigger actions as soon as a message is received. The model we propose is built on the service concept.

**Services.** When a request sent by a client agent is received and accepted by the recipient agent, the execution of the corresponding service is initiated by the recipient agent which then acts as a provider agent. In SaGE, a service is a reified concept; it is executed in its own thread and it defines an execution context. The execution of services can explicitly be controlled by their own agents (for example, to interrupt them). Figure 2 shows how services are created in the SaGE system.

These services fall into two categories:

- **atomic services**, the execution of which does not depend on other services. For example, in the travel agency study case, the *Get price* service (see Fig. 1) that returns a *Provider’s* bid is atomic because it needs no subservice.

- **complex services**, the execution of which requires other services to be achieved. For example, in the travel agency study case, the *Organize a travel* service (see Fig. 1) which handles a *Client’s* initial request is complex as it sends requests that trigger additional service executions.

---

**Fig. 1.** Execution resulting from a request to a travel agency
public class Broker extends SaGEAgent
{
    Service s = new Service ("Search for a travel", getAddress(), serviceID)
    {
        public void live ()
        {
            // body of the service
        }
        public SaGEException concert (Vector subServicesInfo)
        {
            // body of the exception resolution function
        }
        public void handle (NetworkException exc)
        {
            // handler for NetworkException
        }
        public void handle (NoProviderException exc)
        {
            // handler for NoProviderException
        }
    }
}

Fig. 2. Service definition and association of handlers to services in SaGE

The nature of these two kinds of services is very different. An atomic service can be implemented by a simple thread that executes the corresponding treatment and, then, sends back to its client the corresponding answer. A complex service has to be implemented by an entity that is able to send requests and receive responses. Consequently, implementing complex services as agents is necessary and natural, as complex services need the same communication capabilities and cooperative concurrency support as those of agents. Cascaded requests result in cascaded service executions that form a logical structure (tree) of execution contexts. Figure 1 shows the graph of execution contexts that results from the cascaded requests in the travel agency example. This structure is comparable to the call-stack in procedural or object-oriented programming: it provides an explicit representation of both the individual (atomic services) and collective (complex services) activities of agents that enables the management of cooperative concurrency.

Every time a provider agent receives and accepts a request, it logs the ID of the client agent for which it is executing the corresponding service. Accordingly, every time a client agent successfully sends a request, it logs the id of its demanding service, the id of the request, and the id of the provider agent. This log is used to return responses, propagate exceptions and manage the termination of subservices.
Managing Collective Requests. Section 2.2 introduced the concept of role which allows the broadcasting of messages to a set of agents that share a common ability. In order to manage such collective requests, the execution model has to be extended with entities that represent and manage roles. As for services, such entities must be able to send and receive asynchronous messages. In our proposition, we choose to consider these entities as dedicated agents called **role agents** that:

- maintain a list of its participating agents (those which play the corresponding role),
- define a generic treatment for the received requests that consist in broadcasting messages to all its member agents,
- and, collect answers and exceptions from its member agents and combine them into a pertinent collective response or concerted exception.

The execution model described in this subsection provides a means to coordinate the activities of agents as illustrated by the example of Fig. 1. In this example, *Broker* is an agent, *ProviderRole* is a role agent to which three *Provider* agents have subscribed. This execution model is used to integrate the exception handling system presented below.

4.2 An Exception Handling System Dedicated to MASs

Definition of Handlers. In SaGE, exception handlers can be associated with services, agents or role agents.

1. **Handlers associated with a service** are designed to catch and treat exceptions that are raised, either directly or indirectly, while executing the service. This enables a precise, contextual, definition of handlers: the objective of the service, its current state and the impact of exceptions on its completion can be taken into account when coding the handler.

2. **Handlers associated with an agent** are a practical means to define a single handler for all the services of this agent at a time. For example, the death of an agent or the coherence maintenance of agent-specific data can be dealt with such exception handlers.

3. **Handlers associated with a role** are designed to treat exceptions that concern all agents which play a given role. For example, when exceptions occur during the handling of a broadcasted request, partial results or QoS statistics can be returned by handlers at the role level.

These handlers are distinct from the handlers provided by the underlying implementation language. They are triggered by the signaling or the propagation of an agent-level exception. A handler is classically defined by the set of exception types it can catch and by an associated treatment (as illustrated by Fig. 2). SaGE provides a termination model that allows a handler to:
execute some treatments, in order to manage the consequences of the abrupt 
interruption of a service execution, such as restoring the agent in some co-
herent state, sending some partial results, etc.,

- send in turn an exception that signals that it has not been able to successfully 
  manage the exception,
- re-launch a complete execution when associated with a service, after having 
  possibly modified the execution context in order to re-try to successfully 
  achieve it.

**Exception Signaling.** Agent-level exceptions are signaled, during the exe-
cution of services (or their associated handlers), thanks to calls to a specific 
primitive (see Fig. 3). Both exception systems are compatible: language level 
exceptions, caught by language-level handlers, can be turned into agent-level ex-
ceptions by calling the SaGE exception signaling primitive within the language-
level handlers. As other exception signaling primitives, ours takes the exception 
to be signaled as a parameter. A call to this function internally triggers the 
exception handling mechanism of SaGE.

```java
signal (new SaGEException("Bad client address", getOwnerAddress ());
```

**Fig. 3.** Exception signaling in SaGE

**Handler Search.** The heart of an exception handling system is the way han-
dlers are searched for. When an exception is signaled, the execution of the defe-
cutive service is suspended. First, a handler for this type of exception is searched 
for locally, i.e. in the list of handlers associated with the service. If such a han-
der is found, it is executed. If not, the search carries on among the handlers 
associated with the agent that executed the defective service. In all the above 
cases, the defective service is terminated.

If no adequate handler is found in the previous step, it means that the service 
failed and that the consequences of its failure must be dealt with by the client. 
Thus, the exception is propagated to the client agent that forwards it to its 
concerned service. The search carries on there, first in the service itself and, 
then, in the agent for which it executes. This client agent can either be an agent 
or a role agent.

If no handler is found, the search process iterates (the whole process is illus-
trated in Fig. 4) until an adequate handler is found or the top-level is reached. 
In the latter case, the whole computation is aborted.

**Concerted Exception Support.** Inspired by [23] and [24], SaGE integrates 
concerted exception support in its exception propagation mechanism. This mecha-
nism allows:

- not to react to under-critical situations,
- to collect exceptions to reflect a collective or a global defect.
The concerted exception mechanism is available both at the service and the role level. No concerted exception handling is required at the agent level. Indeed, the association of handlers to agents is provided as a facility to define handlers that are common to all the services of the agent. As such, they are managed as handlers associated with services. Thus, the exceptions that trigger these handlers are concerted at the service level by the exception resolution functions associated with the services.

**Concerted Exception Support at the Service Level.** An exception that is propagated to a service is not always critical for the service completion. Indeed, a request can be redundantly sent to several agents to increase reliability and performance. This is, for example, the case when a Provider-RoleAgent sends \( n \) requests to \( n \) Provider agents: the failure of few providers is not critical. In such a case, only the failure of a significant proportion of the requested service providers might be critical. This example is illustrated on Fig. 5.

To enable concerted exception support, propagated exceptions are not directly handled by the recipient service. Such exceptions are stored in a log which is associated to the recipient service. This log maintains the history of the so far propagated exceptions (along with information such as the sources of the exceptions). Whenever a new propagated exception is logged, the concerted exception function associated to the recipient service is executed to evaluate the situation. This function acts as both a filter and a composition function. Depending on the nature or the number of logged exceptions, this function determines if an exception is to be effectively propagated. If so, the propagated exception can be the last propagated one (in case it is critical enough) or a new exception that is calculated from a set of logged exceptions, the conjunction of which creates a critical situation (represented by a concerted exception).

As for handlers, resolution functions are associated with services and each resolution function is specific to a service. To write such functions (see Fig. 5), programmers have access to the exception log – in order to decide if an exception
is to be propagated – and to the exception signaling primitive – to effectively signal the chosen concerted exception.

Concerted Exception Support at the Role Level. The set of requests emitted by a role agent to manage a collective request is transparent for the client agent that sends a request to a role agent. The role agent acts as a collector for responses and sends back a single (composite) response to its client. A comparable scheme is used to concert exceptions. Whenever an exception is propagated from an agent belonging to the role, the exception resolution function, associated to the role agent, is invoked. It logs the exception and, when the cumulative effects of the under critical exceptions becomes critical, it computes the concerted exception to be effectively propagated to reflect the actual global situation.

In the travel agency example, there are cases where concerted exceptions are required. The Poll Providers service of the Broker agent broadcasts a request to get prices from Provider agents. None of these requests is individually critical. Thus, the exception resolution function (see Fig. 5) associated with the ProviderRole_RoleAgent role agent will collect the exceptions signaling the failures of Provider agents without signaling any exception until a critical proportion of these agents fails. There are also cases where individual exceptions

Fig. 5. Exception resolution function associated to the TravelProviders_RoleAgent role agents

```java
public SaGEException concert (Vector subServicesInfo)
{
    int failed = 0;
    int pending = 0;

    // count the number of exceptions raised in subservices and the number of
    // subservices that are still running
    for (int i=0; j<subServicesInfo.size (); i++)
    {
        if (((ServiceInfo) (subServicesInfo.elementAt (i)).getRaisedException () != null)
            failed ++;
        else ((ServiceInfo) (subServicesInfo.elementAt (i)).isFinished () == false)
            pending ++;
    }

    // if more than 30% failed, there are two many bad providers
    if (failed > (0.3*subServicesInfo.size ()))
        return new SaGEException("too_many_bad_providers",getAddress ());

    // if not, at the end, only few providers failed
    if (failed != 0 && pending == 0)
        return new SaGEException("few_bad_providers",getAddress ());

    // computing still running - no critical situation
    return null;
}
```
are critical. Services like *Select an offer* or *Contact parties* are critical for the successful completion of the *Search for a travel* service: their failure immediately results in the failure of the client service (the concerted exception function associated with the *Search for a travel* service does not filter exceptions propagated from the *Select an offer* or *Contact parties* services nor delay their handling).

## 5 Overview of the Implementation of Sage for MadKit

For this first implementation of the Sage model, we did not modify the kernel of the MadKit platform but choose to specialize classes from its core implementation (*AbstractAgent*, *Agent*, *ACLMessage*) along with the standard Java *Exception* class (see Fig. 6).

![Class Diagram of the Sage model](image)

**Fig. 6.** Class Diagram of the Sage model

### 5.1 Communication

In MadKit, agents are referenced by their logical *AgentAddress* addresses. These addresses are used to route *ACLMessage* messages to the recipient agents through the middleware. The class *ACLMessage* has been specialized in order to encapsulate data which is specific to our execution model:

- identifiers used to manage the internal message forwarding from agents to their services,
- standard definition of message categories (such as *request*, *finish*, *terminate*, *exception*) [25].
5.2 Exceptions

We have extended the standard Java Exception class in order to:

- differentiate agent-level exceptions from language-level ones,
- encapsulate information which may be useful for programmers such as the address of the agent which signals or propagates the exception,
- remain compatible with the standard Java EHS (agent level exceptions can thus be, if required, simply considered as standard Java exception and finally caught and treated by classical Java handlers).

5.3 Agents and Services

The AbstractAgent class is the base class of the communicating entities in the system: every AbstractAgent instance has an address, can send messages to another AbstractAgent and read the received messages in its message box. The Agent class is the base class for all the active entities in the system. It extends the AbstractAgent class by adding the live method, the method that is first executed after an agent is created and a thread is attached to it by the kernel of the system. This method is to be overridden in subclasses in order to define the main behavior of the agents in a given execution model. The live method in the SaGEAgent class implements a loop that actively scans the message box of the agent. When a message is received, it is handled by the handleSaGEMessage method that calls more specific methods depending on the category of the message (request, exception, etc.). Though they are active entities too, the base class of services, Service, is not implemented as a subclass of Agent, not to be mistaken with agents (services are internal entities, encapsulated in agents). However, Service is a subclass of AbstractAgent in order to inherit of the same communication capabilities as agents\(^2\).

The SubServiceInfo class is used by services to reference the services they request along with management data such as their significance and their execution status.

5.4 Broadcasting

Roles agents (introduced in Sect. 4.1) are implemented as a specialized class of SaGEAgent agents by the RoleAgent class. They handle the broadcasting of the requests they receive with the generic broadcastService method. It is to be noticed that the RoleAgent class is defined in such a way that defining a role in a SAGE application only implies defining which role the corresponding RoleAgent manages (see Fig. 7) and, optionally, associating handlers and a dedicated resolution function with it (see Fig. 5).

\(^2\) In the current implementation, we do not differentiate atomic services from complex services.
public class TrainProviders_RoleAgent extends TravelProviders_RoleAgent {
    public TrainProvider_RoleAgent () {
        super("train-provider");
    }
}

Fig. 7. Definition of the Train-Providers RoleAgent

Fig. 8. A typical MADKit window with launched SaGEAgents

6 Experimentation

After having implemented SaGE, we experimented it with the travel agency example (see Fig. 8).

6.1 Definition of the Agents of the Travel Agency Example

In order to implement this example with SaGE, we had to implement three agents (Client, Broker and Provider) and two role agents which handle collective requests for both transport means (see Fig. 9): the TrainProviders_RoleAgent role agent and the PlaneProviders_RoleAgent role agent.
In addition, in order to allow the debugging of SaGE agents, we provide two extra agents (see Fig. 8):

- the Logger, which logs into a text file actions related to services (instantiation, initiation, termination, exception signaling, etc.),
- and, the ServiceTracer which visually and dynamically represents the service tree, the root of which is the service initiated by the last launched Client agent (see Fig. 10).

6.2 Concerting Exceptions

In the travel agency example, concerting the exceptions propagated from Provider agents at the role level allows pertinent actions to be performed. If few providers signal exceptions, partial results may still be send to the client as shown in Fig. 10(a). On the contrary, if too much Provider agents signal exceptions, an exception is to be propagated to the client in order to notify it of a global problem as shown in Fig. 10(b). This behavior is implemented in the exception resolution function (see Fig. 5) and the handler (see Fig. 11) associated with TravelProviders_RoleAgents role agents.

6.3 Termination

A service which terminates its execution (either standardly or exceptionally) forces the termination of all its pending subservices (see Sect. 4.1). For example, in Fig. 10(b), a concerted exception is propagated up to the root of the service tree. Each service which propagates the concerted exception has exceptionally finished. In particular, the handle_ticket_request service, during its exceptional termination, forces the termination of its init_contract pending subservice as it becomes useless. The logical tree structure that is formed by the cascaded
service requests is thus used upward to manage the propagation of exceptions to dependant services and downward to manage the termination of pending useless requested services.

7 Conclusion and Future Work

In this paper we propose an original exception handling system for MASs. It distinguishes itself from previous work because it does not rely on the use of entities external to agents but fully integrates exception handling mechanisms to the execution model of the agents. It allows in-context, pertinent handlers to be defined that can directly be associated with the services provided by an agent, as part of its behavior. The execution model supports cooperative concurrency and manages the propagation of exceptions between cooperating agents. Moreover, individual exceptions propagated from agents that contribute to a collective activity can be concerted into more pertinent exceptions regarding the management of those global activities. Handlers and exception resolution
public void broadcasthandle (SaGEException exc, BroadcastService bs) {
    if (exc.getMessage ().equals ("few_bad_providers"))
        {  
            bs.setalive (true);
            bs.broadcastavailableresponses ();
        }
    else
        {  
            sendMessage (bs.getparentowneraddress (), new SaGEMessage ("exception",
                        exc.getMessage (), bs.getrequestID ()));
            bs.terminate ();
        }
}

Fig. 11. Handler associated to the TravelProviders_RoleAgent role agents

functions can be associated with different kind of execution model entities (services, agents, roles) in order to support exception handling in different contexts (from the local behavior of agents to the collective activities in roles, through one-to-one collaborations between agents).

We implemented and successfully experimented this model: the experimentation is available as an applet\(^3\).

Various perspectives are considered, such as to extend our EHS in order to be able to resume the execution of a service to some chosen point after the successful treatment of an exception. Another perspective is to transpose SaGE to component-based platforms (such as J2EE/JMS technologies) [26] or to other message-oriented middlewares (WebServices).

Acknowledgments

The authors thank Jacques Ferber, creator of the MADKit system [1,16] for his contribution to this work, and for many profitable discussions.

References


\(^3\) http://www.lgi2p.ema.fr/~fsouchon/sage_applet/sage_applet.html
8. Anonymous: Rationale for the design of the ada programming language. ACM SIGPLAN Notices 14 (1979) 1–139


On Manageability and Robustness of Open Multi-agent Systems

Naftaly H. Minsky and Takahiro Murata

Department of Computer Science
Rutgers University, Piscataway, NJ, 08854 USA
{minsky, murata}@cs.rutgers.edu

Abstract. This paper addresses the manageability of large, open and distributed multi-agent systems (MASs). Specifically, we are concerned here with three critical elements of management: (a) the ability of the manager to monitor relevant operations of its subordinates; (b) its ability to steer its subordinates; and (c) and the robustness of the agent-community under certain unexpected adverse conditions, such as unpredictable failure of the manager itself.

Using our own law-governed interaction (LGI) mechanism, we show how one can build a scalable infrastructure that supports monitoring, steering and a degree of robustness, in spite of the heterogeneous and dynamic nature of the system being managed, and in spite of possible changes and failures of the manager itself.

1 Introduction

Among the conditions underlying multi-agent systems (MASs), or multi-agent communities, none is more critical than the ability of the disparate autonomous agents of such a community to coordinate their activities. Coordination is, of course, indispensable for effective cooperation between agents, as well as for safe competition between them. A flock of birds, for example, must coordinate its flight in order to stay in formation; and car drivers must coordinate their passage through an intersection, if they are to survive this experience. Every such coordination is based on a certain policy, i.e., a set of rules of engagement – such as stopping at a red light, in the driving case – that must be complied with by all community members, for the community to operate harmoniously and safely.

The implementation of a coordination policy is straightforward when dealing with a close-knit community, whose members can all be carefully constructed to satisfy the policy at hand. This is how birds of a feather flock together, having their rules of engagement inborn. This is also how the processes spawned by a single program coordinate their activities, by having their rules of engagement built in, by the programmer, or by the compiler.

But the implementation of such policies is much more problematic when dealing with large scale and open agent-communities, whose members are heterogeneous – possibly written in different languages, and constructed by different
organizations, without any knowledge of each other – and whose membership may change dynamically. Such open communities are increasingly common over the Internet. Consider, for example, the agents of disparate institutions collaborating under a grid agreement [4]; or a distributed committee of people [18] who need to deliberate, and make communal decisions about certain issues, subject to some version of the Robert’s Rules of Order.

In a previous work [15] by one of the authors (Minsky) it has been argued that in order to ensure that all members of such an open community conform to a given coordination policy, this policy needs to be stated explicitly – and not be embedded in the code of the disparate agents – and that it must be enforced. Moreover, we have argued that for large scale communities, such enforcement of coordination policies needs to be decentralized, to be scalable. This thesis has been the guiding principles behind the design [13] and implementation [15] of the coordination and control mechanism called law-governed interaction (LGI). This mechanism has already been used to support collaborative work, like the decision making of a distributed committee [18]; and to enable safe competitive interactions, via the imposition of access control policies.

In this paper we address an important mode of coordination: managerial coordination\(^1\), where one (or more) agent is managing certain operations of other members of the community. Specifically, we will be concerned here with three critical elements of management:

**Monitoring.** The ability of the manager to monitor relevant operations of its subordinates, so that he, she, or it\(^2\) knows the state of whatever is being managed.

**Steering.** The ability of the manager to change the course of actions of its subordinates, in response to the dynamically changing state of the activity being managed.

**Robustness.** The ability of the community in question to recover from some unexpected adverse conditions, such as unpredictable failure of the manager itself. (Such an ability is also known as fault tolerance, and self-healing.)

The rest of this paper is organized as follows. We start in Sec. 2 with an example, involving a team of buyers for a department store, intended to illustrate the nature of management in a distributed MAS, and the difficulties in supporting such management in large and open communities whose individual components may be unreliable. Sec. 3 is an overview of LGI, which provides the computational foundation for this paper. We show how LGI can be used to support management of multi-agent systems by applying it to our buying team

---

1 For a reader who doubts that there is any relationship between coordination and management we note that Malone and Crowston defined coordination as “the managing of dependencies between agents in order to foster harmonious interaction between them” [11].

2 Since the problem and the solution we present in this paper apply to both software agents and to human agents operating with some software interface, we henceforth use “it” in referring to an agent.
example, as follows: in Sec. 4 we show how the buying team can be monitored and steered by its manager, even though the identity of the manager itself may change dynamically, but in an orderly fashion; and in Sec. 5 we show how an unpredictable failure of the team’s manager can be recovered from. We discuss related work in Sec. 6, and we conclude in Sec. 7.

2 On the Nature of Management in Open Multi-agent Systems: An Example

We will use here an example, to help illustrate the nature and the difficulties of management in open multi-agent systems, and the kind of policies that can support such management.

Consider a department store that deploys a team of agents, whose purpose is to supply the store with the merchandise it needs. The team consists of a manager, and a set of employees (or the software agents representing them) authorized as buyers, via a purchasing-budget provided to them.

Let us suppose that under normal circumstances, the proper operation of this buying team would be ensured if all its members comply with the following, informally stated, policy, called BT.

1. For an agent to participate in the buying team, it must authenticate itself as an employee of the department store via a certificate issued by a specific certification authority (CA), called here admin.
2. The buying team is initially managed by a distinguished agent called firstMgr. But any manager of this team can appoint another agent authenticated as an employee as its successor, at any time, thus losing its own managerial powers. The appointment of a new manager must be published via a designated publish/subscribe (P/S) server [16] called psMediator, and every buyer, i.e., a team member with a purchasing budget, is required to subscribe to such publications.
3. A buyer is allowed to issue purchase orders (POs), taking the cost of each PO out of its own budget – which is thus reduced accordingly – provided that the budget is large enough. The copy of each PO issued must be sent to the current manager.
4. An employee can be assigned a budget by the manager, and it can give some of his budget to other employees, recursively. Also, the manager can reduce the budget of any employee e, as it sees fit, which freezes the budget of e, preventing others from increasing e’s budget.

Discussion: First note the open nature of this community. A buyer can be any agent properly authenticated by the CA admin, and is supposed to operate autonomously in deciding what to buy, at which price, and from whom – but they are limited by their purchasing budget. Also, the budget may be assigned either by the manager, or by anybody with a budget, obtained directly or indirectly from the manager. Moreover, the manager itself can be changed in the middle of the purchasing activity, as specified in Point 2 of this policy.
In spite of the openness of the buying team, it would be manageable – that is, its manager would be able to monitor and steer it – if this policy is complied with by all team members. In particular monitoring is provided by Point 3 of this policy, which requires a copy of every PO to be sent to the manager. And each buyer should know the identity of the current manager, because by Point 2, all buyers are required to subscribe to the announcement of management transfer. (Note however, that there is a possible race condition here, when the manager is replaced, which is not handled by this simplified policy, but is fully handled by its treatment under LGI, in Sec. 4.) Also, the manager should be able to steer the buying team, by sending messages to various agents, adjusting their budget as it sees fit.

But how can one be sure that all members of an heterogeneous and dynamically changing community will conform to this policy? In particular, how can one be sure that all buyers will report to their manager about every PO they issue; that they would limit themselves to the budget available to them; or that they will obey their manager’s message intended to reduce their budget? We maintain that such assurances can be provided only if some policy like $BT$ is enforced. In Sec. 4 we show how this can be done, scalably, under LGI.

Finally, we will show in Sec. 5 how one can ensure the recovery of this buying team from a failure of its own manager, or from a breakdown of communication between the manager and its team members.

3 Law Governed Interaction (LGI) – An Overview

LGI [13] is a message-exchange mechanism that allows an open group of distributed agents to engage in a mode of interaction governed by an explicitly specified policy, called the interaction-law (or simply the “law”) of the group. The messages thus exchanged under a given law $L$ are called $L$-messages, and the group of agents interacting via $L$-messages is called an $L$-community $C_L$ (or, simply, a community $C$).

By agents, referring to participants of an $L$-community, we mean autonomous actors that can interact with each other, and with their environment\(^3\). An agent might be an encapsulated software entity, with its own state and thread of control, or a human that interacts with the system via some interface; in either case, no assumptions are made about its structure and behavior. A community under LGI is open in the sense that its membership can change dynamically, and can be very large, and its members can be heterogeneous. For more details about LGI than provided by this overview, the reader is referred to[15,3].

3.1 On the Nature of LGI Laws

and Their Decentralized Enforcement

The function of an LGI law $L$ is to regulate the exchange of $L$-messages between members of a community $C_L$. Such regulation may involve (a) restriction

\(^3\)A similar notion of agents, particularly with their societal aspect, has also been adopted by others, including [20].
of the kind of messages that can be exchanged between various members of $C_L$, which is the traditional function of access-control; (b) transformation of certain messages, possibly rerouting them to different destinations; and (c) causing certain messages to be emitted spontaneously, under specified circumstances, via a mechanism we call obligations.

A crucial feature of LGI is that its laws can be stateful. That is, a law $L$ can be sensitive to some function of the history of the interaction among members of $C_L$, called the control-state ($CS$) of the community. The dependency of this control-state on the history of interaction is defined by the law $L$ itself.

But the most salient and unconventional aspects of LGI laws are their strictly local formulation, and the decentralized nature of their enforcement. This architectural decision is based on the observation that a centralized mechanism to enforce interaction-laws in distributed systems is inherently unscalable, as it can become a bottleneck, and a dangerous single point of failure. The replication of such an enforcement mechanism, as seen in the Tivoli system [9], would not scale either, due to the required synchronous update of $CS$ at all the replicas, when dealing with stateful policies.

The Local Nature of LGI Laws: An LGI law is defined over a certain types of events occurring at members of a community $C$ subject to it, mandating the effect that any such event should have. Such a mandate is called the ruling of the law for the given event. The events subject to laws, called regulated events, include (among others): the sending and the arrival of an $L$-message; the coming due of an obligation; and the occurrence of an exception in executing an operation in the ruling for another event. The agent at which a regulated event has occurred is called the home agent of the event. The ruling for a given regulated event is computed based on the local control state $CS_x$ of the home agent $x$ – where $CS_x$ is some function, defined by law $L$, of the history of communication between $x$ and the rest of the $L$-community. The operations that can be included in the ruling for a given regulated event, called primitive operations, are all local with respect to the home agent. They include: operations on the control-state of the home agent, such as insertion (+t), removal (-t), and replacement (t<-s) of terms; operations on messages, such as forward and deliver; and the imposition of an obligation on the home agent.

To summarize, an LGI law satisfies the following locality properties: (a) a law can regulate explicitly only local events at individual home agents; (b) the ruling for an event e can depend only on e itself, and on the local control-state $CS_x$ of the home agent $x$; and (c) the ruling for an event can mandate only local operations to be carried out at the home agent $x$.

Decentralization of Law-Enforcement: The enforcement of a given law is carried out by a distributed set $\{T_x \mid x \in C\}$ of controllers, one for each member of community $C$. Structurally, all these controllers are generic, with the same law-enforcer, and all must be trusted to interpret correctly any law they might operate under. When serving members of community $C_L$, however, they all carry the same law $L$. And each controller $T_x$ associated with an agent $x$ of this community carries only the local control-state $CS_x$ of $x$, while every $L$-message ex-
Fig. 1. Enforcement of the law

changed between a pair of agents $x$ and $y$ passes through a pair of controllers, $\mathcal{T}_x$ and $\mathcal{T}_y$ (see Fig. 1).

Due to the local nature of LGI laws, each controller $\mathcal{T}_x$ can handle events that occur at its client $x$ strictly locally, with no explicit dependency on anything that might be happening with other members in the community. It should also be pointed out that controller $\mathcal{T}_x$ handles the events at $x$ strictly sequentially, in the order of their occurrence, and atomically. These greatly simplify the structure of the controllers, making them easier to use as our trusted computing base (TCB).

3.2 The Implementation Status

The LGI mechanism has been implemented (in Java) as a messaging middleware called Moses\textsuperscript{4}. For writing laws, LGI currently supports two languages: (a) a Prolog-like language, introduced in [13], employed in the rest of the paper, and (b) a restricted version of Java. The most recent version of the controller evaluates a single regulated-event in less than 100$\mu$s, on a Sun Ultra-10 (440 MHz), when a law is written in (b) above.

3.3 The Deployment of LGI

As will be explained in Sec. 3.4, the trust between the members is immune to the manner in which the text of the law is made available. Thus, having located a controller via a controller-service described in Sec. 3.2, an agent supplies this controller with the law $\mathcal{L}$ it wants to employ, by specifying the text of $\mathcal{L}$ or its URL. After checking that law $\mathcal{L}$ is well-formed, the controller starts to serve this client. Only through this hand-shake between a controller and an agent – a procedure called adoption of law $\mathcal{L}$ – the agent can start to participate in $\mathcal{L}$-community $\mathcal{C}_L$. All these kinds of communication between an agent and its controller, including ones mentioned below, are facilitated by a Moses API, while a graphical user interface is provided for human users.

\textsuperscript{4} A public distribution version is being finalized as of the time of writing.
Once $x$ has adopted law $\mathcal{L}$, it may need to distinguish itself as playing a certain role, etc., which would provide it with some distinct privileges under law $\mathcal{L}$. This can be done by presenting certain digital certificates to the controller, where law $\mathcal{L}$ specifies a trusted CA, and the kinds of attributes that need to be certified in such certificates. A simple illustration of such certification is provided by our example law $\mathcal{BT}$ in Sec. 4, under which one may claim to be an employee of the department store, or to possess the name of the designated manager, for example.

3.4 The Basis for Trust between Members of a Community

Note that we do not propose to coerce any agent to exchange $\mathcal{L}$-messages under any given law $\mathcal{L}$. The role of enforcement here is merely to ensure that any exchange of $\mathcal{L}$-messages, once undertaken, conforms to law $\mathcal{L}$. In particular, the enforcement mechanism ensures that a message received under law $\mathcal{L}$ has been sent under the same law; i.e., that it is not possible to forge $\mathcal{L}$-messages. As described in [3], this is assured by the following: (a) The exchange of $\mathcal{L}$-messages is mediated by correctly implemented controllers, certified by a CA specified by law $\mathcal{L}$; (b) these controllers are interpreting the same law $\mathcal{L}$, identified by a one-way hash [17] $H$ of law $\mathcal{L}$; and (c) $\mathcal{L}$-messages are transmitted over cryptographically secured channels between such controllers. Consequently, how each member $x$ gets the text of law $\mathcal{L}$ is irrelevant to the assurance that all members of $\mathcal{C}_L$ operate under the same law.

Finally, note that although we do not compel anybody to operate under any particular law, or to use LGI, for that matter, one may be effectively compelled to exchange $\mathcal{L}$-messages, if one needs to communicate with others that operate only under this law. For instance, given law $\mathcal{BT}$ in Sec. 4, which governs the buying team, introduced in Sec. 2, if the manager is committed to interact via $\mathcal{BT}$-messages, then other employees will have to operate under law $\mathcal{BT}$ as well, if they are to become buyers, i.e., with any budget at all for purchases. This is probably the best one can do in the distributed context, where it is impossible to ensure that all relevant messages are mediated by a reference monitor, or by any set of such monitors.

4 Providing for Monitoring and Steering of a Buying Team

We are now in a position to establish a managerial infrastructure over the team of buyers introduced in Sec. 2. We start, in this section, by formalizing policy $\mathcal{BT}$ via an LGI law, thus providing for the monitoring and steering of our buyers team, under normal circumstances. In the following section we present a modification of this law, which helps the team to recover from the failure of its manager. Both laws have been fully implemented and tested.

Law $\mathcal{BT}$, displayed in its entirety in Figs. 2 and 3, consists of two parts: called the preamble and the body. The preamble of $\mathcal{BT}$ contains the following clauses:
Preamble:
  cAuthority(pk1).
  authority(admin, pkAdmin).
  alias(firstMgr, "firstMgr@host-a.somestore.com").
  alias(psMediator, "psMediator@host-b.somestore.com").

R1. adopted(_) :- Self=firstMgr, do(+manager(1)), do(imposeObligation(o, 600)).
  When entering the community, firstMgr gets its status token.

R2. certified([issuer(admin), subject(Self), attributes([employee])]) :- do(+employee).
  An employee is required to present a certificate issued by CA admin.

R3. sent(X, giveBudget(A), B) :- A>0,
    (manager(V)@CS, M=mgr(X, V)
     ;mgr(Y, V)@CS, M=mgr(Y, V), budget(D)@CS, D>=A,
       do(decr(budget(D), A)),
       do(forward(X, giveBudget(A,M), B))).

R4. arrived(X, giveBudget(A, mgr(M, V)), B) :- employee@CS,
    (mgr(M1, V1)@CS->V>V1, do(mgr(M1, V1)<-mgr(M, V)),
     do(undelivered(L)<-undelivered([])), cleanUp(M, L)
     ;do(+mgr(M, V)),
     (frozen@CS, do(forward(B, returnBudget(A), X))
      ;(budget(D)@CS, do incr(budget(D), A)); do(+budget(A)),
        do(forward(Self, subscribe(newMgr([], psMediator)))))),
    do(deliver)).
  A manager, as well as a buyer, can give some budget to an employee.

R5. sent(M, removeBudget(A), B) :- manager(_)@CS, A>0, do(forward).

R6. arrived(M, removeBudget(A), B) :- budget(D)@CS, do(+frozen),
    (D>=A, R=A; R=D), do(decr(budget(D), R)), do(deliver).
  A manager can remove some budget from a buyer.

R7. arrived(B, returnBudget(A), X)
    :- (budget(D)@CS, do incr(budget(D), A)); true), do(deliver).
  Any returned budget is added back.

Fig. 2. Law $B^T$ of buying team

First there is the cAuthority(pk1) clause that identifies the public key of the certification authority (CA) to be used for the authentication of the controllers that are to mediate $B^T$-messages. This CA is an important element of the trust between the agents that exchange such messages. Second, there is an authority clause, which provides the public key of the CA that would be acceptable under this law for certifying employees, which are to be allowed to participate in this
8. \( \text{sent}(B, \text{po}(S,A), V) \) :- mgr(M)@CS, budget(D)@CS, A\( \geq 0 \), D\( > A \), do(decr(budget(D),A)), do(deliver(B,Msg,V)), do(forward(B,po(S,A,V),M)). 

9. \( \text{arrived}(B, \text{po}(S,A,V), M) \) :- (manager(\_)@CS, do(deliver); do(forward(M,notMgr(Msg),B))). 

A buyer can issue a PO, within its budget, which is monitored by the manager.

10. \( \text{sent}(M, \text{transfer}(V), N) \) :- manager(V)@CS, do(-manager(V)), do(forward), do(repealObligation(o)). 

11. \( \text{arrived}(M, \text{transfer}(V), N) \) :- (employee@CS, W is V+1, do(+manager(W)), do(deliver), do(forward(N, publish(newMgr(id(N),ver(W))), psMediator)), do(imposeObligation(o,600)) ; do(forward(N,noTrans(V),M))). 

12. \( \text{arrived}(N, \text{noTrans}(V), M) \) :- do(+manager(V)), do(deliver), do(imposeObligation(o,600)). 

The manager can transfer its power to one of the buyers.

13. \( \text{oilationDue}(o) \) :- manager(V)@CS, Self=M, do(forward(M, publish(newMgr(id(M),ver(V))), psMediator)), do(imposeObligation(o,600)). 

When obligation \( o \) comes due, the manager’s info is published.

14. \( \text{arrived}(A, R, psMediator) \) :- do(deliver). 

15. \( \text{sent}(psMediator, N, A) \) :- do(forward). 

Messages at the P/S server receive no further ado.

16. \( \text{arrived}(psMediator, \text{notify}(\text{newMgr}(id(M),ver(V))), A) \) :- mgr(M1,V1)@CS, V>V1, do(mgr(M1,V1)<-mgr(M,V)), do(undelivered(L)<-undelivered([L])), cleanUp(M,L). 

When the notice of the new manager is received by a buyer, the id of this manager is stored in the control-state, replacing a previous one, if any.

17. \( \text{arrived}(M, \text{notMgr}(PO), B) \) :- undelivered(L)@CS, do(undelivered(L)<-undelivered([PO|L])). 

18. cleanUp(\_,[]). 
19. cleanUp(M,[PO|R]) :- do(forward(Self,PO,M)), cleanUp(M,R). 

An undelivered PO copy is kept, and sent to the new manager.

Fig. 3. Law \( B^T \) of buying team (cont’d)

buying mission; this CA is given a local name, admin in this case, to be used within this law. Finally, alias clauses provide shorthand for the identifiers (ids) of two specific agents: firstMgr and psMediator.

The body of this law is a list of all its rules, each often followed by a comment (in italic), which, together with our discussion, should be understandable even for a reader not well versed in our language for writing laws. Each rule has a head, to the left of symbol :-, and a body, to its right. Recall that the same
law is interpreted individually by the controller associated to each agent in the community. A regulated event occurring at this home agent triggers a rule that has a matching head, if any (the matching is done in the order in which the rules are written). The triggered rule proceeds to check if all the goals in its body are attained, given the control-state of this agent.

In addition to the standard types of Prolog goals, the body of a rule may contain two distinguished types of goals. These are the sensor-goals, to “sense” the control-state of the home agent, and the do-goals that contribute to the ruling of the law. A sensor-goal has the form \texttt{t@CS}, where \texttt{t} is any Prolog term. It attempts to unify \texttt{t} with each term in the control-state of the home agent. (A variant of this, \texttt{t@L}, does the same for an arbitrary list \texttt{L}.) A do-goal, which always succeeds, has the form \texttt{do(p)}, where \texttt{p} is one of the primitive-operations, mentioned in Sec. 3.1. It appends the term \texttt{p} to the ruling of the law. Thus, successful evaluation of a rule body with do-goals leads to a non-empty ruling, and the execution of the primitive operations therein. In what follows, we may speak of this effect as if the said rule itself were to execute the pertinent operations. (By default, an empty ruling implies that the event in question has no consequences – such an event is effectively ignored.)

Before we present the specific aspects of law \( BT \), some general remarks are in order. First, terms in each agent’s control-state are used to represent the role played by this agent. In particular, the control-state of the current manager should contain a term, \texttt{manager(V)}, while this manager remains in its power, where \( V \) stands for this manager’s “version number” – which starts with 1 for the initial manager, \texttt{firstMgr} (see Point 2 of policy \( BT \)), incremented by 1 for each new manager thereafter. Likewise, the presence of term \texttt{budget(D)} in the control-state of employee \( x \) means that \( x \) has a budget of amount \( D \), and is entitled to act as a buyer. At the same time, \( x \)’s control-state should have a term, \texttt{mgr(M,V')} , signifying that the most recent manager known to \( x \) is \( M \), under its version number \( V' \), and thus used to send copies of POs issued by \( x \). (Note, due to a race condition, such \( V \) and \( V' \) may not always coincide, whose handling shall be explained below.)

Second, a publish/subscribe (P/S) server, \texttt{psMediator} is used as the means for announcing the appointment of a new manager. To ensure that these announcements would actually be conveyed to each buyer acting at the time, law \( BT \) ensures that an employee subscribes to such announcement when, and only when, it becomes a buyer (i.e., by receiving some budget). Moreover, an acting manager is forced to announce its identity periodically, for the purpose to be clarified later.

We now describe law \( BT \) in detail by focusing on the following aspects of it: (a) establishing roles; (b) monitoring and steering of purchasing activities; (c) manager change; and (d) handling of race conditions. We will end this section with some further discussion.

Establishing Roles: First, policy \( BT \), in its point 1, requires participants to be certified as an employee by \texttt{admin}, the designated CA. The presentation of a valid certificate by the subject of the certification (i.e., a self-certificate) leads to
triggering rule $\mathcal{R}2$, where a specific term, employee, is inserted to the control-state. Second, the control-state of firstMgr, distinguished by its name in Point 2, is initialized by rule $\mathcal{R}1$ with its designated term manager(1), when it joins the community. The space limit prevents us from showing the following: (a) how to ensure psMediator is available when firstMgr effectively starts acting; and (b) how to prevent firstMgr from entering the community more than once throughout its history.

Finally, in contrast to rather "static" roles seen above, the role of a buyer is established through dynamic interaction of the agents involved: that is, when an employee receives a budget for the first time, by rule $\mathcal{R}4$, it will have a term, budget(A), mentioned above.

**Monitoring and Steering of Purchasing Activity:** Policy BT (in its Point 4) allows the manager to steer the purchasing activities by adjusting the budget held by the buyers. This provision is implemented by rules $\mathcal{R}3$ through $\mathcal{R}6$, which allow the manager to send a message of the form of either giveBudget(A) or removeBudget(A) to an employee, resulting in increasing or decreasing, respectively, the recipient’s budget by amount A. Note that the budget freezing provision is implemented as follows: term frozen is inserted to the control-state of a buyer whose budget is being decreased ($\mathcal{R}6$), and the presence of this term prevents any budget from being added ($\mathcal{R}4$); in such a case, the forwarded budget is returned, and added back, as applicable ($\mathcal{R}7$). Rules $\mathcal{R}3$ and $\mathcal{R}4$ also allow a buyer to transfer its budget to another employee, as stipulated in the same policy point.

By Rule $\mathcal{R}3$, when a manager sends a budget to some agent, a term of the form $\text{mgr}(M,V)$ is attached to the message, informing the buyer of the current identity and version of its manager; this term will replace the manager information, of the same form, at the recipient, if it turns out to carry a more recent version number. This manager information is used when a buyer issues a PO, by rule $\mathcal{R}8$ (Point 3), in order to address a copy of the PO to the manager. If the recipient is in fact the current manager, by rule $\mathcal{R}9$, such a copy is delivered, fulfilling the monitoring requirement. The other case will be explained as part of the handling of race conditions.

Note that by $\mathcal{R}8$ an issued PO (bound to the $\text{Msg}$ built-in variable) is delivered remotely, causing no regulated event, at the chosen vendor, which is not expected to be a member of this community. Such a message is possibly signed by the controller, using its private key, allowing the recipient to verify the authenticity of the message.

**Manager Change:** According to Point 2 of policy BT, by rule $\mathcal{R}4$ a subscription of the form, newMgr([],), is sent out to psMediator on behalf of an employee receiving a budget for the first time, and is delivered by rule $\mathcal{R}14$. Thus, every buyer is bound to have a subscription to every newMgr event-notice.

---

5 Syntax $(P;Q)$ in the laws should read $P$ or $Q$; similarly $(P\rightarrow Q;R)$ means if $P$ then $Q$ else $R$.

6 This is really necessary for an agent that gets its first budget, thus becoming a buyer.
When the manager wants to transfer its power to another member, according to Point 2, it sends a `transfer` message with its version number, handled by rule $\mathcal{R}10$, where the `manager` term is removed, taking away its privileges. If the forwarded `transfer` message arrives at an employee, by rule $\mathcal{R}11$, the recipient becomes the new manager under the succeeding version number, $\mathcal{W}$, while a `newMgr` event-notice, carrying $\mathcal{W}$, is published via `psMediator` ($\mathcal{R}14$ and $\mathcal{R}15$), and disseminated to each subscriber of this kind of notices ($\mathcal{R}16$). (If the recipient of the `transfer` message is not an employee, a `noTrans` message is returned by rule $\mathcal{R}11$, and processed by rule $\mathcal{R}12$, to reinstate the old manager.) Thus, a buyer, whose subscription to such notices has been processed by `psMediator` in time, gets notified of this transition, leading to an (possible) update of the recent manager information. Cases where some buyers do not get such notices (at least not in time for reporting POs) are explained immediately below.

**Handling of Race Conditions:** As seen in the implementation of the monitoring and that of the manager change above, the consequence of race conditions arises as potential staleness of the manager information, kept in term $\text{mgr}(M, V)$ at every buyer, to be used for forwarding a PO copy. Such a condition occurs, for example, if the arrival of the `newMgr` notice is too late, or is entirely missed.

Law $\mathcal{BT}$ has the following safety measures against such stale information: When a PO copy arrives, unless the recipient is the current manager, a `notMgr` message is returned, wrapping the copy ($\mathcal{R}9$), which is appended to the list argument of term `undelivered` at the buyer ($\mathcal{R}17$). In addition, the law provides three complementary ways to refresh stale manager information: (a) through the event-notice of the manager transition ($\mathcal{R}16$); (b) through additional budget, carrying the most recent manager information known to the sender ($\mathcal{R}4$); and (c) through periodic announcement of the current manager, carried out via an `obligation` named $\mathcal{o}$, which comes due, in these cases every 10 minutes ($\mathcal{R}13$), as long as the manager is in power ($\mathcal{R}1$, $\mathcal{R}10$, $\mathcal{R}11$, and $\mathcal{R}12$). Once the manager information is refreshed at a buyer carrying undelivered PO copies as above, their delivery to the new manager is attempted ($\mathcal{R}18$ and $\mathcal{R}19$). Note that the staleness of this refreshed information would be handled in the same manner as described here.

**Discussion:** A comment about our use of P/S for dissemination of global information is in order. First, we could have relied on “gossip” mechanism among the buyers to disseminate the manager information. We chose P/S instead because its greater predictability. Second, we did not use the P/S server to pass copies of POs to the manager, since we are concerned with information security. That is, communication via P/S would be more vulnerable than via (possibly encrypted) unicast channel, and is less preferred in dealing with highly sensitive information, such as POs.

5 Recovering from Unexpected Failures of a Manager

Although law $\mathcal{BT}$ allows for the manager to be changed dynamically, while the team being managed is operating, such changes are to be well organized: the
current manager appointing its successor. In this section we will build an infrastructure that would allow our buying team to recover from an unpredictable failure of its manager.

More specifically, we will consider two kinds of failures: (a) the failure of the manager itself; and (b) the failure of the LGI-controller (i.e., part of our own infrastructure) that serves the manager, mediating all interaction between the manager and members of its team. In both cases we assume the failure to be of a fail-stop kind. We also assume that the underlying network, and the rest of our infrastructure, including the P/S servicer, does not fail. For a broader perspective on such treatment of failures, as part of work on self-healing, see [12].

Every failure recovery mechanism must have two elements: the detection of the failure, and its handling. For the handling of failures we adopt here the notion of guardian originally proposed by Tripathi et al. [19]7. That is, we assume that there exist an agent called guardian, which, when notified that the current manager failed, would appoint another employee to this post. Part of our contribution here is that we ensure – via a law described below – that the designated guardian will have the power to appoint new managers, that he would be the only one who has that power, and that the various buyers would obey the new managers, thus appointed by the guardian – all this despite the openness of the community in question. Such assurances are, of course, critical to the effective recovery of the team, and its proper operation.

The other critical part of our contribution is that we can ensure that a failure would, in fact, be detected, and that the designated guardian will be duly notified of it. These assurances are provided differently for failures of types (a) and (b) above, by law BT′ as we shall see below.

Law BT′, a modification of our original law BT, is defined in Figs. 4 and 5. It contains all of BT, including the preamble and the rules – some of which are modified and are labeled with primed numbers in these figures – and adds six new rules. The new parts of the modified rules appear in bold letters.

Law BT′ adds guardian as a designated agent in its preamble; the state of this guardian is initialized by R1′. Among others, this law ensures that the guardian is always informed of the current manager and its version number, representing this information via a term currentMgr(Id,V) in its control-state. This law is discussed in some details below.

Handling of Failure of Kind (a): This is the case where the manager itself fails, but its controller does not. This would allow us to employ this controller, as governed by law BT′, to detect the failure of the manager. For this purpose, we assume that the manager is supposed to acknowledge (by being programmed or otherwise) the receipt of every copy of a PO delivered to it. Therefore, in the absence of such a timely acknowledgement, the manager is determined to have failed by its controller. Such detection works as follows under law BT′:

First, whenever a copy of a PO arrives at the controller of the manager, rule R9′ imposes an obligation, marked with that PO copy, which will become due in 60 seconds. This obligation would be repealed, by rule R20, when the

---

7 A similar concept has been studied by Klein et al. [10].
**Preamble:**

\[ \text{alias}(\text{guardian}, \text{guardian}@\text{host-x.somestore.com}). \]

**R1'.** \( \text{adopted}(.): (\ldots; \text{Self}=\text{guardian} \rightarrow \text{do}(+\text{currentMgr}(\text{firstMgr}, 1))) \).

**R9'.** \( \text{arrived}(B, \text{po}(S,A,V), M) \)

\[
\begin{align*}
&: - (\text{manager}(.)@\text{CS}, \text{do}(\text{deliver}), \\
& \quad \text{do} (\text{imposeObligation} (\text{po}(S,A,V), 60)) \\
& \quad ; \text{do}(\text{forward}(M, \text{notMgr}(\text{Msg}), B)).
\end{align*}
\]

When the manager receives a PO copy, an obligation is imposed.

**R20.** \( \text{sent}(M, \text{ack}(\text{PO}), .) : - \text{PO} = \text{po}(S,A,V), \text{do}(\text{repealObligation}(\text{PO})). \)

When the manager acknowledges the receipt of a PO copy, the corresponding obligation is repealed.

**R21.** \( \text{obligationDue}(\text{po}(S,A, .)) : - \text{manager}(V)@\text{CS}, \text{do} (-\text{manager}(V)), \\
\text{do}(\text{forward}(\text{Self}, \text{mgrFailed}(V), \text{guardian})). \)

If the receipt of a PO copy is not acknowledged timely, the corresponding obligation comes due, which takes away the power of the manager, and sends a message to this effect to guardian.

**R22.** \( \text{arrived}(M, \text{mgrFailed}(V), \text{guardian}) : - \text{do} (-\text{currentMgr}(., .)), \\
\text{do}(+\text{readyToAppoint}(V)), \text{do}(\text{deliver}(\text{Self}, \text{appoint}, \text{Self})). \)

A report of a failed (current) manager leads to having guardian appoint a new manager of its choice.

**R23.** \( \text{sent}(\text{guardian}, \text{appoint}(B), .) \\
\quad : - \text{readyToAppoint}(V)@\text{CS}, \text{do} (-\text{readyToAppoint}(V)), V_1 \text{ is } V+1, \\
\quad \text{do}(+\text{currentMgr}(B, V_1)), \text{do}(\text{forward}(\text{Self}, \text{transfer}(V), B)). \)

Fig. 4. Law \( \text{BT}' \) to handle failures at the manager.

Manager acknowledges the receipt of the PO copy, within 60 seconds. Otherwise, the obligation comes due, and triggers rule \( R21 \), which removes the term, \( \text{manager}(V) \), from the control state, effectively depriving the manager of its power, and sends a \text{mgrFailed} message to \text{guardian}. Once this message arrives at \text{guardian}, by rule \( R22 \), the \text{currentMgr} term that corresponds to this manager is removed from the control state, and a prompt to appoint a new manager is delivered to \text{guardian}; this state of \text{guardian} is characterized by the presence of term \text{readyToAppoint}(V) with \( V \) bound to the version of the current, failed manager.

Now, when \text{guardian} sends in a message to appoint some member as the new manager, by rule \( R23 \), a new \text{currentMgr} term for this appointee is inserted, while a \text{transfer} message is sent to the appointee, and the manager’s role is transferred, as seen in Sec. 4, if the appointee is a legitimate employee. Otherwise, rule \( R12' \) allows \text{guardian} to try appointing another member, by removing the \text{currentMgr} term for this appointee, and re-inserting the \text{readyToAppoint} term.
A message to transfer the manager’s role is now routed through guardian.

If forwarding of a PO copy fails (due to the failure of the controller associated with the manager), a report to this effect is sent to guardian.

The Revision of the Orderly Management Transfer: As opposed to rule R10 of law BT allowing the current manager to send a transfer message directly to a member, N, of its choice, to appoint N as its successor, rule R10’ forwards the message to guardian. Rule R24 reflects the change of the manager in a proper currentMgr term at guardian before forwarding the transfer message to N. Notice that a noTrans message to signal illegitimacy of the appointee would be sent back to guardian. Thus, in order to give the retiring manager, M, a chance to adjust the choice of its successor, a term byMgr(B,M) is inserted into the control-state of guardian by R24, and used by R12’ to propagate the message back to M.

Handling of Failure of Kind (b): If the controller associated with the current manager fails, any message forwarded to it, particularly the copy of a PO issued by a buyer, causes a regulated event of type exception at its sender. Such an exception is handled by rule R25, which adds the PO copy to the list of undelivered copies. Moreover if this exception has happened with the current manager known to this buyer, a mgrFailed message is sent to guardian, which is handled in the same manner as in the case of failure of kind (a) above.

Finally, other failures can be dealt with adequately as well, but we only outline some of them for brevity: (a) a failed member or its controller that is supposed to receive a transfer (or noTrans) message, sent by guardian above, should be handled similarly to the above; (b) PO copies that do not receive acknowledgement from the manager can be accumulated in its control-state, and sent to guardian upon the determination of the manager’s failure, for further
delivery to the new manager to be appointed; and (c) a failure of guardian or its controller can be addressed by replicating the service of guardian, which would not sacrifice the scalability of the MAS very much, considering the failures at managers and those at such guardians are relatively rare.

6 Related Work

There has been a growing interest in coordination in recent years, and a variety of different, and quite powerful, coordination mechanisms have been devised. We provide here a short overview of these mechanisms, and conclude with relevant work on exception handling in MAS.

Most current coordination mechanisms have been devised for close-knit groups, generally written in a single language, and spawned by a single program. This is certainly true for Linda [6], one of the first, and most influential, explicit attempts at coordination. Linda features very powerful and elegant coordination primitives, but it provides for no explicit statement of a policy. So, if a group of agents are to coordinate effectively via a tuple-space (a basic Linda concept) they all need to internalize some kind of common policy. (Many Linda-based mechanisms, like Sonia [5], for example, share this property of Linda.) Of course, no such internalization can be relied on in an open group – which is what concerns us in this paper.

There is a collection of mechanisms that do provide for an explicit coordination policy, but only for groups of agents spawned by a single program or, at least, written in a specific language. These techniques are not applicable to open groups, in our sense of this term, where the group members may have been written independently, possibly in different languages. Some of these mechanisms, including Actors [1], Contracts [8], and Composition Filters [2], do not depend on centralized control and are thus, potentially scalable.

Perhaps the closest work to this paper is programmable tuple-space [7], called LuCe. Like us, they call for an explicit formulation of a policy, which is to be written in a formal language. The programmability is achieved by triggering a reaction whenever a communication event occurs. A reaction, similar to our rule, consists of a set of primitive operations to be executed when the event occurs. A major difference between LGI and LuCe is that the latter is based on the fundamentally centralized Linda model, while LGI is entirely decentralized. A decentralized treatment, under LGI, of interaction via Linda tuplespaces, which pushes much of the control to the client-side, is described in [14].

Regarding exception handling, or the treatment of failures, in MASs: Tripathi et al. [19] introduces the notion of guardian, which is a dedicated agent that consolidates the handling of exceptions that other “ordinary” agents do not deal with. A similar notion has been proposed by Klein et al. [10]. This is an important concept which we adopted, adding to it several critical elements, such as: (a) the formal, and enforced, specification of what powers should such a guardian have, and who could serve as such a guardian; and (b) the assurances that a guardian would be invoked when the exceptions in question happen. A broader perspective from the point of view of self-healing is presented in [12].
7 Conclusion

The issue discussed in this paper is the management of large, open, and distributed multi-agent systems, where by “open” we mean that the system consists of an heterogeneous group of agents, whose membership may change dynamically, in an unpredictable fashion. We are particularly concerned with two critical elements of management: (a) the ability of the manager to monitor relevant operations of its subordinates; and (b) the ability of the manager to steer its subordinates.

Using our own law-governed interaction (LGI) mechanism, we have shown how one can build a scalable infrastructure that provides for both monitoring and steering, in spite of the heterogeneous and dynamic nature of the system being managed, and in spite of possible changes of the identity of the manager itself. Moreover, we have shown how such a system can recover from some unexpected failures, such as a failure of the manager itself.

Although this management infrastructure has been designed specifically for our buying team example, many of its elements are quite general, and it can therefore be applied to a wide range of multi-agent systems.

References


Security Mechanisms for Mobile Agent Platforms Based on SPKI/SDSI Chains of Trust

Michelle S. Wangham, Joni da Silva Fraga, Rafael R. Obelheiro, Galeno A. Jung, and Elizabeth Fernandes

Department of Automation and Systems Federal University of Santa Catarina C. P. 476 – 88040-900 – Florianópolis – SC – Brazil {wangham,fraga,rro,galeno,elizabeth}@das.ufsc.br

Abstract. This work defines a security scheme, based on SPKI/SDSI chains of trust, for protecting mobile agent platforms in large-scale distributed systems. The scheme is composed by a protocol of mutual authentication, a mobile agent authenticator and a mechanism for the generation of protection domain. Due to the flexibility of the SPKI/SDSI certificate delegation infrastructures used, the proposed scheme provides a decentralized control for authorization and authentication.

1 Introduction

New techniques, languages and paradigms have appeared to facilitate the creation of distributed applications in several areas. Perhaps the most promising paradigm is the one which incorporates the mobile agent concept. A mobile agent in a large-scale network can be seen as a software agent that travels through a heterogeneous network, crossing various security domains and executing autonomously in its destination.

The migration of an agent in a distributed infrastructure is characterized by the transference of its code and state [1]. In this context, the state of the agent is its execution state and agent attribute values which determine what must be done when the execution is resumed in the destination computing environment. The use of mobile agents implies a need for extra services on the machines where those codes will execute. Each machine needs a environment – an agent platform – whose purpose is to support the applications to relocate dynamically its software components in different sites. This platform is used to create, interpret, transfer and resume agents. The platform where an owner creates an agent is called its home platform, and is a trusted environment for the agent.

In spite of its advantages, widespread adoption of the mobile agent paradigm is being delayed due to security concerns. Currently available mechanisms for reducing the risks of this technology do not efficiently cover all the existing threats, besides introducing performance restrictions that frequently outweigh the benefits from the use of mobile agents.
Due to the characteristics of the mobile agent paradigm and the threats to which it is exposed, security mechanisms must be designed to protect the communication support, agent platforms and the agents themselves. This paper focuses on the issue of security in agent platforms, considering large-scale systems.

In face of the possible attacks and failures of mobile agents, the resources and functionalities of agent platforms should be protected. Establishing isolated execution domains (protection domains) for each incoming agent and controlling access to the system domain is an approach that has been commonly adopted with the purpose of offering this protection. Protection against malicious agents is not restricted to confining their execution to their own execution domains in agent platforms. Other issues need to be considered when distributed large-scale systems are the focus. For instance, the generation of these protection domains depends on the authentication and the privilege attributes (credentials) of agents. Thus, when these large-scale systems are considered, the implementation of authentication and authorization mechanisms is a great challenge.

In order to protect against an agent, a platform depends not only on the verification of the agent’s owner authenticity, but also on the degree of trust in the platforms already visited by the agent, since a mobile agent can become malicious by virtue of its state having been corrupted by previously visited platforms [2]. Ensuring reliability and security of multi-hop agents with free itineraries is not an easy task and there is no effective solution for that, since its destination and the visited platforms are not previously known.

Based on decentralized authentication and authorization controls that use certificate SPKI/SDSI delegation mechanisms [3], this work defines a scheme for protecting agent platforms against malicious mobile agents for large-scale systems. This scheme includes prevention techniques (mutual authentication of platforms and a technique for generating protection domains) and a malicious agent detection technique (authentication of multi-hop mobile agents). In our scheme, the flexibility needed to meet specific application requirements is given by the ability to select only the subset of available security mechanisms desired by each application.

2 Security in Mobile Agent Platforms

Mobile agent platforms face several threats, such as [4]: masquerading, when an agent poses as an authorized agent in an effort to gain access to services and resources to which it is not entitled; denial of service, when an agent launch attacks to consume an excessive amount of the agent platform’s computing resources; and unauthorized access, for example when an agent obtains read or write access to data for which it has no authorization.

Establishing isolated domains for agents is the most common technique for protecting agent platforms resources and functionalities against malicious agents. In addition to this approach, other techniques have been proposed based on conventional security techniques. Some of these techniques are:

- **Safe Code Interpretation.** The main idea is to execute commands considered harmful in a safe place or deny them to the agent ([5] and [6]).
- **Digital Signatures.** This technique works as a way to confirm the authenticity of an agent, its origin and integrity. Typically, the code’s signer is the agent’s creator. Since an agent works on behalf of an organization or a end user, mobile agent platforms (e.g. Ajanta [7] and SOMA⁠[¹]) usually take the user’s signature as an indication of the authority.

- **Path Histories** [8]. The main idea of this technique is to keep a verifiable record of the platforms already visited by an agent, so that the next platform to be visited can determine if the agent will be processed and which resource restrictions will be applied.

- **Proof-Carrying Code** (PCC) [9]. This technique follows an approach based on code verifiers and associates portions of an agent’s code to formal proofs that assure code correctness and that are verified before the code execution. In this approach the code producer (e.g. the author of an agent) must provide proofs that the program has safety properties previously arranged with the code consumers.

These mechanisms offer an effective security to agent platforms and their resources for some classes of applications, particularly when techniques are combined. For instance, the domain isolation technique combined to code signing (provided in the Java 2 platform) makes it possible to implement run-time access control based on the authority of an agent (the owner)². However, these techniques are not suitable to large-scale applications. Moreover, access control based only on the owner of an agent does not seem to be appropriate when multi-hop agents with free destinations are taken into consideration, since the trust in an agent depends not only on the owner and the forwarding platform but also on all platforms visited by the agent [8].

Therefore, an effective authentication and authorization scheme for a large-scale system should be based on privilege attributes (credentials), on the identity of the principal responsible for the agent and on the authenticity of the platforms visited by the agent. In this work we use authorization certificates to represent agent credentials. This certificates are signed statements of authorization that bind a capability to a cryptographic key instead of binding a name to a key.

### 3 SPKI/SDSI Infrastructure

The complexity of X.509’s global naming scheme motivated two related although independent initiatives: Simple Public Key Infrastructure (SPKI), an IETF RFC initiative [3], proposed by Carl Ellison and others, and Simple Distributed Security Infrastructure (SDSI), designed at MIT by Ronald Rivest and Butler Lampson [10]. These proposals were later merged in a single specification called SPKI/SDSI.

SPKI/SDSI uses an egalitarian model of trust. Subjects (or principals) are public keys and each public key is a certification entity [11]. There is no global

¹ http://lia.deis.unibo.it/Research/SOMA/main.shtml

² Limitations of the Java 2 access control model are described in section 7.
hierarchical infrastructure as in X.509. Two types of certificates are defined by
SPKI/SDSI: name and authorization certificates. A name certificate links names
to public keys, or even to other names. The concatenation of the public key of
the certificate issuer with a local name represents a SPKI/SDSI unique global
identifier. A certificate is always signed with the private key of the issuer. An
authorization certificate grants specific authorizations from the issuer to the
subject of the certificate; these authorizations are bound to a name, a group
of names or a key. The issuer of the certificate can also allow a principal to
delegate the received permissions to other principals. This delegation allows a
controlled distribution of authorization. An SPKI authorization certificate has
the following fields: the issuer’s key; the subject of the certificate; the delegation
bit (Boolean); a tag to specify an authorization; and a validity period for the
certificate.

The SPKI/SDSI delegation model allows the construction of chains of trust
that begin with a service manager and arrive in principals’ keys. When a given
subject desires to obtain access to some resource, it must present a signed request
and a chain of authorization certificates for checking the needed permissions [11].
One of the main purposes of this checking is to guarantee that even if the issuer
at the top of the chain grants the right of subsequent delegation, no subject can
obtain authorizations higher than the original issuer’s ones.

SPKI/SDSI has been broadly establishing itself as a support platform for au-
thentication and authorization in distributed systems. It follows a completely de-
centralized approach to authentication and authorization control. This approach
appears to be well-suited for implementing security controls in large-scale dis-
tributed systems. Due to these advantages, authorization certificates were chosen
to represent agent credentials in our scheme.

4 Security Scheme Based on Chains of Trust

The security mechanisms proposed here are based on a model of agents that
assumes free itineraries and multi-hop. The Mobile Agent Facility (MAF) spec-
ification [1] is used as a guideline to interoperability between agent systems.
The scheme proposed here has a completely decentralized authentication and
authorization control that uses SPKI/SDSI chain of trust and the concept of
federations.

4.1 Platform Federations

In this scheme, mobile agent platforms can group according to their service
classes, constituting service federations and defining trust relationships among
themselves. For example, the Mold Enterprise Federation may group the mobile
agent platforms from enterprises that manufacture molds and matrices. The con-
cept of federations, introduced in [12], aims to facilitate the access of clients to
services through dynamic establishment of chains of trust. These chains of trust
between client and service are quickly and efficiently established from SPKI
name and authorization certificates available at the certificate repository for the
service federation. Besides, a member of a federation can join other federations
and different federations can establish trust relationships, creating webs of federations with global scope. The main function of a web of federations is to help a client, through its agents, in the search for access privileges that link it to the guardian of a service (another platform).

Federations must provide certificate repositories and support for certificate chain discovery. By storing name and authorization certificates in these repositories, the services available in the associated platforms can be announced. The inclusion of a platform in one of these federations should be negotiated with the Certificates Manager (CM) that controls this certificate storage service. Further details on the concept of federations can be found in [12].

4.2 Authentication and Authorization Scheme

During the mobile agent creation process, the owner, being the authority that an agent represents, provides a set of SPKI/SDSI authorization certificates defining agent’s privilege attributes (its credentials). A list that indicates service federations whose member platforms are authorized to execute the agent can be (optionally) defined and attached to the agent. The owner of the agent has to put in an object that will contain the list of previously visited platforms (called the path register) a signature indicating its identity and the identity of the first platform to be visited. This object is attached to the agent. The path register and the list of federations are used to analyze the history of the agent’s travels in the agent authentication process. Finally, the agent’s owner must sign the code of the agent and its read-only data, and then create the agent in its home platform to send it through the network.

Fig. 1 shows the procedures defined in the security scheme, composed by prevention and detection techniques that emphasize the protection of agent platforms and their resources. In this scheme, the source and destination platforms (the ones which send and receive agents, respectively) first go through a mutual authentication protocol so that a secure channel between them can be established. After that, the agent will be sent with its credentials to be authenticated by the destination platform and then its domain of protection can be created. In other words, when an agent arrives in a platform it presents its public key and a chain of SPKI/SDSI certificates to a verifier that performs authorization checks. From these information, this verifier must generate the permissions required for the agent to be run in a protection domain on its destination platform. The particularity of the generation of the set of permissions required for agent execution in a destination platform makes the scheme flexible and follows the principle of least privilege [13]. Chains of trust also help to achieve the necessary scalability for Internet-based applications. The following section analyzes some aspects referring to the mechanisms in the proposed scheme.

Mutual Authentication of Platforms. According to the MAF specification [1], agent platforms should use authentication and authorization mechanisms to grant or deny accesses to their resources prior to receiving agents or interacting
with other agents or platforms. In the proposed scheme, mutual authentication between the platforms involved must be established before agents can be transferred. This authentication creates a secure channel in the communications infrastructure between the authenticated parties that is used for agent transfer.

In accordance to the SPKI/SDSI model, identification is not done with names, but with public keys, with digital signatures as the authentication mechanism. Thus, in platform authentication, for a digital signature to be verified, a public key and a digital signature verification element must be present at the receiver. In the SPKI/SDSI egalitarian model, the public keys for checking signatures are found in chains of authorization presented by the source platforms to the destination ones.

In our proposal, mutual authentication is performed during the secure channel establishment between agents platforms. Fig. 2 shows the mutual authentication performed with a challenge/response protocol, based on SPKI/SDSI certificates of the owners (managers) of the platforms. The basis for authentication in SPKI/SDSI are chains of authorization certificate [3].

In step 1, in Fig. 2, the source platform sends a signed message containing a request (establish\_trust) and a nonce\textsuperscript{3} (nonceSP), without any certificates. From this request, the destination platform builds a signed challenge and sends it to the source platform so that it can prove it has the required permissions (step 2). The challenge is composed by information from the resource’s ACL, by nonceSP and by a nonce generated by the destination platform (nonceDP). In step 3, the source platform verifies the signature of the challenge to confirm the authenticity of the destination platform. Then, it sends a signed response with the request, nonceDP, and the authorization certificates for the desired operation. From the chain of certificates, the destination platform can check the requester’s signature, finishing the authentication process (step 4).

The support required for implementing this protocol is provided by the SPKI/SDSI Resolver object which is available in all platforms. It is also through this object that the SPKI/SDSI infrastructure is available in each platform.

It is important to note that the process of mutual authentication of platforms is concluded with the establishment of a secure channel. This channel will be used for all agents that are transferred between the two platforms, without need for subsequent platform authentication.

\textsuperscript{3} The purpose of this nonce is to protect from replay attacks.
Mobile Agents Authentication. Before instantiating a thread to an agent, destination platforms must authenticate received agents. Digital signatures cannot be used for authentication of mobile agents due to a restriction that prevents agents from carrying their private keys. One of contributions of this paper is the definition of a multi-hop authenticator that establishes trust on an agent based on the authenticity of the owner of the agent, on the authenticity of the platforms visited and on the federations defined by the owner of the agent.

Consider the multi-hop algorithm as currently defined is shown in Fig. 3; upon receiving a mobile agent, the platform must first check, through verification of the agent’s signature (which covers its code and read-only data), that this agent has not been corrupted and confirm its association to a principal, its owner (step 1). Thus, modifications introduced by malicious platforms are detected by any platform visited by the agent. When a platform detects a modification, it should warn the home platform so that countermeasures may be taken.

For one-hop agents, the technique proposed in step 1 ensures the integrity of an agent, but for multi-hop agents this technique is insufficient. For detecting possible modifications and checking the agent’s travelling history, the destination agent platform must analyze the agent’s path register (step 2). For that purpose, each platform visited by an agent should add to the agent’s path register a signed entry containing its identity (public key) and the identity of the next platform (public key) to be visited, forming the history of the path followed by the agent. In step 2, Fig. 3, the platform administrator has to define how the agent’s path register is analyzed and how the trust level is established. The administrator reviews the list of visited platforms (verifying if they are associated in some federation indicate in the agent’s federations list) and/or verify the authenticity of each entry in the agent’s path register.\(^4\)

\(^4\) An inconvenient is that analyzing an agent’s path becomes costlier as the path register grows.
Moreover, we propose that platform-generated sensitive data (write once data) are stored in a secure repository of partial results named $Worepository$, which is carried by an agent (Figure 4). To securely insert a partial result into the repository, an agent platform must follow a cryptographic protocol, as proposed by Karjoth et al. in their KAG family of protocols [14]. In the KAG family of protocols, a mobile agent visits a sequence of platforms $P = P_1, P_2, \ldots, P_n$, departing from $P_0$ (its home platform) and returning to $P_{n+1} = P_0$. This approach builds a chain of result encapsulations chaining each result entry to all previous entries end to the identity of the next visited platform. Each platform digitally signs its entry using its private key and uses a secure hash function to chain the partial results encapsulated by previous platforms and the identity of the next platform. Based on Yee’s forward integrity property [15], the authors define the strong forward property. This property guaranteed that, for $k < m$ (where $m$ is the last visited platform), no partial result encapsulation $R_k$ obtained from a platform $P_k$ can be modified without this modification being detected [14].

The scheme we propose supports protocols P1 (Publicly Verifiable Chained Digital Signature Protocol) and P2 (Chained Digital Signature Protocol with Forward Privacy) from the KAG family of protocols. An agent’s owner must choose the most appropriate protocol to insert results into the repository according to the desirable security properties of each application. Information about which protocol is being used is contained in the agent’s credentials. Table 1 presents the security goals of protocols P1 and P2 [14].

As discussed by their authors, protocols from the KAG family have a limitation: the partial result encapsulation $R_k$ from the last visited platform must not be modified. If this result is modified, the strong forward integrity and insertion resilience properties no longer can be guaranteed. Moreover, if two or more malicious platforms conspire, they might control all data established between the first and the last conspirator on the itinerary. Two platforms conspire if they exchange secrets used in building the chaining relation. The same situation arises
Table 1. Security Goals of Protocols P1 and P2

<table>
<thead>
<tr>
<th>Goals</th>
<th>Protocol P1</th>
<th>Protocol P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Confidentiality</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Non-repudiability</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Strong Forward Integrity</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Publicly Verifiable Integrity</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Insertion Resilience</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Forward Privacy</td>
<td>no</td>
<td>yes</td>
</tr>
</tbody>
</table>

![Agent Diagram](image)

**Fig. 4.** Structure of a Mobile Agent

when an agent visits the same platform twice [14]. It is our intention to address the protection of agents in depth in future work. Fig. 4 shows the structure of a mobile agent in our scheme.

**Procedure for Generation of Protection Domains.** Protection domains and the permissions assigned to them are defined after trust in an agent has been established (a result from previous procedures). They are based on the agent’s SPKI/SDSI authorization certificates and on trust and risk levels. The authorization chains carried by an agent, representing its credentials, need to be verified by the platform guardian for the set of permissions to be defined. This scheme decouples the privilege attributes granted by principals (agent’s credentials) from the attributes required to access resources protected by the platform (the access control policy), offering a more flexible, dynamic and suitable access control for large-scale systems.

It should be noted that the initial authorization certificates associated to the agent during its creation may not be sufficient to grant access to certain resources in a given platform. So, new certificates can be provided to the agent during its visits to other agent platforms. For example, considering the travel agency case study, suppose that an agent needs to visit a hotel associated to an airline. The agent may not have the certificates needed to be received by this platform. Thus, the platform that represents the airline can delegate certificates
to this agent enabling it to access the associated hotel. The trust model proposed by SPKI/SDSI determines that a client is responsible for finding certificates that enable it to access a given service. Therefore, an agent can search for certificates on the webs of federations and negotiate them when they are found.

5 Implementation of the Proposed Scheme

A prototype of the security scheme for protection of agent platforms has been implemented in order to demonstrate its suitability for distributed applications with mobile agents (see Fig. 5).

![Fig. 5. Architecture of the Prototype](image)

For the mobile agents support layer we have chosen IBM Aglets\(^5\), an open-source platform that uses Java as its mobile code language. Besides being considered a de facto standard for programming distributed applications, Java has several properties that make it a good language for mobile agent programming, such as: portability, safe code interpretation, multithreaded programming, object serialization, dynamic code loading, and code signing. However, although Java is very convenient for the creating agents, its static and centralized access control model poses some limitations with regard to security policy definition.

The Aglets software development kit (ASDK) provides mechanisms for code and state information mobility, and an environment (called Tahiti) which supports creation, cloning, execution, dispatching, and retraction of agents. Aglets supports both Agent Transfer protocol (ATP) and Java Remote Method Invocation (RMI) as communication infrastructures. We have, in our research group, extended this platform to also support CORBA, but these extension are not going to be discussed in this paper.

The SPKI/SDSI Infrastructure component of the prototype architecture is responsible for the creation of SPKI/SDSI certificates for agent platforms and for mobile agents. A Java library that implements SDSI 2.0 [16] and that provides these functionalities is being used for this purpose. We have extended

\(^5\) http://aglets.sourceforge.net/
this library to provide increased flexibility and efficiency to agent platform. The SDSI 2.0 library is also used in the implementation of SPKI/SDSI federations in our schemes. The repositories of certificates are implemented using the Apache Xindice\(^6\) native XML database. Certificates are stored in XML format in order to ease searches for creation of new chains.

The Common Object Request Broker Architecture (CORBA) standard [17] has been chosen for the middleware layer. CORBA was selected due to its great acceptance (it is also an ISO standard) and for providing an object-oriented architecture that follows open systems concepts and defines standards for heterogeneous environments, allowing the construction of applications that are independent of programming language, operating system, hardware and network technology. The middleware layer enables the mobile agents to interact with stationary agents which can be written in any programming language. This interaction is required by the application described in section 6.

The implementation of the security scheme shown in Fig. 1 is discussed below.

5.1 Mutual Authentication

The protocol for mutual authentication (see Fig. 3) was implemented with the SDSI 2.0 library and the Java 2 cryptographic tools. Due to limitations to Aglets platform, the protocol for mutual authentication was implemented as a mobile agent (Authenticator Agent) that is activated whenever an agent platform attempts to communicate with another platform. If a secure channel is already established, the communication proceeds as usual. Otherwise, the protocol for establishing mutual authentication between the platforms is started.

As mentioned in section 4.2, SSL (implemented by the iSaSiLk toolkit\(^7\)) is used as an underlying security technology to ensure confidentiality and integrity in the communication channel in our scheme. This interaction with SSL was implemented and integrated to the Aglets platform. The SSL certificates required are generated from the SPKI certificates of the communicating platforms.

5.2 Mobile Agents Authentication

The multi-hop authenticator described in section 3 is being implemented with the Java 2 cryptographic tools and the SDSI 2.0 library. Presently, all entries in the agent’s path register are analyzed considering only the first possibility described in section 3. Only two levels of trust were defined according to the list of federations: authorized or non-authorized platform. That is, the platform either is or is not a member of a federation present in the federations list previously defined for the agent mission.

5.3 Generation of Protection Domains

This stage, which results in the definition of protection domains, has been fully designed but only partially implemented. The process for generating the set of

---

\(^6\) http://xml.apache.org/xindice/

\(^7\) http://jce.iaik.tugraz.at/products/02_isasilk/
permissions was defined to overcome limitations related to the static and centralized approach of Java 2 access control. An agent’s protection domain is generated from the SPKI/SDSI authorization certificates that form its credentials and which are carried by the agent itself.

Some extensions to the Java 2 security model are needed for generating the protection domain where an agent will be run. These extensions, represented by gray ovals in Fig. 6, are: \textit{SPKISecureClassLoader}, required for extracting certificates from the incoming agent and for creating a protection domain of a thread; \textit{SPKIPolicy}, an object that represents a policy that defines, from the certificates carried by an agent, which Java permissions will be associated to the application domain; and \textit{SPKIVerifier}, required for verifying the authenticity SPKI certificates.

![Diagram of Protection Domain Generation](image)

\textbf{Fig. 6.} Dynamics for Protection Domain Generation

Still in Fig. 6, the platform administrator describes the security policy of the agent platform by mapping the authorization granted from SPKI/SDSI certificates to Java permissions, defining for that purpose a policy file. When an agent is received in a platform, its credentials (chain of certificates) are forwarded by \textit{SPKISecureClassLoader} to the \textit{SPKIPolicy} object for interpretation. When the SPKI permissions are mapped to Java permissions, the Java support generates the corresponding protection domain for the thread that runs the agent. Java permissions are available through the \textit{PermissionCollection} object.

If a thread (agent) makes an access request to a resource outside of its application domain, that is, in a system domain, \textit{AccessController} is activated to check whether the access must be allowed. It must verify, in the created protection domain, whether the principal has a corresponding \textit{Permission} object in its collection of permissions. If it does, the thread can exchange domains, entering in the system domain.
6 Integration of the Prototype to a Distributed Application

The prototype presented in section 5 is being used in a mobile agents-based application for searching and selection of partners in the formation of virtual enterprises (VEs) – the Partners Search and Selection (PSS) System described in [18]. In this application, a cluster is basically seen as a group of enterprises strategically grouped by regional or technological similarities to better exploit the market, having the will to collaborate with the others to accomplish a given business opportunity (BO). A Virtual Enterprise (VE) is seen as a temporary alliance of enterprises created around a given business opportunity whose collaborative execution is supported by computer networks. Therefore, in this case, a VE is created from the cluster’s enterprises. The PSS System proposed aims to assign a given business opportunity to the cluster members and to select afterwards the most suitable subset of members – a VE – to carry it out. The matter is not (only) to support the VE creation, but to do so in an agile, smart and secure way [18]. This system explores the benefits of the mobile agents paradigm to improve agility in the presentation of business opportunities to the cluster of companies and to achieve a higher efficiency in the formation and analysis of the possible virtual enterprises to be constituted.

A prototype which implements the PSS system is being developed for the TechMoldes scenario, a group of mold makers in southern Brazil whose members have been collaborating to enhance their global competitiveness. The main idea of Techmoldes is to act as a single/larger productive entity in the market, combining the individual skills and resources of each member, but transparent to the final customer. When collaborating within Techmoldes, each member remains independent and autonomous, even to make business out of the cluster. Techmoldes is currently composed of thirteen companies, including competing companies with equivalent resources.

Three classes of agents compose the PSS system:

- Broker Agent: it is a stationary agent responsible for receiving a business opportunity (BO), distributing it to the potential enterprises, sending an Agent Messenger to them, and collecting/electing the final VE composition.
- Mobile Agent: it is a mobile agent responsible for delivering a BO to the enterprises, negotiating locally with them, and traveling through the net to the other enterprises and finally back to the broker.
- Enterprise Agent: it is a stationary agent responsible for receiving a BO, evaluating it, accessing the local database to get the required information, and answering the BO request to the mobile agent.

The PSS System is shown in Fig. 7. In this system, stationary agents, representing every real company of the virtual organization, are responsible for interactions with the companies’ legacy systems. Mobile agents are used as the mean to travel through the selected enterprises in order to interact with the stationary agents for receiving the required information (e.g., delivery time and capacity) or for negotiating lower costs or shorter delivery times.
When a business opportunity appears, it is received by the broker, which identifies (only) the potential enterprises that can supply each mold (step 1, 7). A summary of the mold specification is immediately sent out to them (step 2). Each enterprise receives it, evaluates its preliminary interest and capacity, and sends back an answer to the broker, either yes (expressing its interest) or no (step 3). The broker receives the answers and sends a mobile agent to the enterprises that answered yes, provided with the full BO specification and the list of enterprises to visit (step 4). The mobile agent arrives at the first enterprise and interacts with the local stationary agent, asking for its delivery time and capacity (step 5). The local agent, acting as the enterprise’s representative, retrieves this information from its legacy system or local database. After this, the mobile agent asks the local supervisor about the price, as it is a very important information in the molding sector. A negotiation process may be carried out locally (step 6). The, the mobile agent moves to the next enterprise of the list with these information (step 7). This process repeats until all the enterprises in the list are visited, when the mobile agent returns to the broker agent with their proposals (step 8). The agent broker generates the set of possible VEs, evaluates every VE composition and a human broker elects the most suitable one, sending a win or lose message to the enterprises afterwards (step 9). The election criteria applied on this case are global lowest cost and shortest delivery time.

The proposed security scheme is being integrated to the PSS System in order to better assess its suitability to Internet-based applications. This integration reinforces the trust building process in the creation of VEs. The security scheme is used to protect the communication channels, the agent platforms, and the agents themselves. To meet the specific requirements of the applications, we are designing a flexible framework that implements the proposed security scheme and allows selecting a subset of mechanisms that could also be adequate to the
applications’ functionalities. For example, if the cluster decides that an enterprise’s answers are sensitive data that cannot be revealed to other enterprises, then the broker can use the framework to configure the scheme so that these data will be encrypted and stored in an agent’s *W*<em>o</em>r<em>e</em>posit<em>o</em>ry.

The mobile agents are coded in Java and the Aglets platform is used. The stationary agents are coded in C++ and the MASSYVE-KIT platform<sup>8</sup> has been used. CORBA is the technology applied to support the multi-platform interoperation.

7 Related Work

Most Java-based agent platforms take advantage of the Java security models (especially the Java 2 version) to implement part of their security mechanisms. Among these platforms, there are commercial ones, such as IBM Aglets and GMDFokus Grasshopper, and academic ones, such as SOMA (University of Bologna) and Ajanta (University of Minnesota).

These mobile agent platforms extend the Java Security Manager to provide a more flexible and adequate solution to agent platforms and to implement protection domains that isolate mobile agents, preventing malicious attacks from them. The difference between authorization schemes in these platforms lies in the information used to determine the set of access rights for an incoming agent. Aglets and Ajanta use only the agent’s owner identity. Grasshopper uses access control policies based on the owner’s identity or on the name of its group (group membership). In SOMA, principals are associated to roles that identify the operations allowed on the resources of the system. All these approaches are not really suitable for large-scale systems due mainly to limitations of the Java 2 access control model.

These limitations stem from the fact that, instead of following the distributed nature of its execution model, the Java 2 security model uses a centralized authorization scheme<sup>9</sup>. When running, each code is labeled as belonging to one or more protection domains. Each domain has a set of permissions associated from a policy configuration file. Therefore, this file defines a static mapping between each mobile component and the permissions granted to it for execution in a local environment. In addition to a number of difficulties related to programming, development of a distributed and dynamic environment is constrained by limitations that stem from the concentration of trust on a single configuration file, which demands an up-front static definition of all distributed components in the system and their corresponding security attributes.

Agent authentication is essential for implementing an effective authorization scheme in mobile agent systems. The Aglets and Grasshopper platforms do not have mechanisms for mobile agent authentication. SOMA authenticates an agent

<sup>8</sup> http://www.gsigma-grucon.ufsc.br/massyve/mkit.htm

<sup>9</sup> This centralization refers to the fact that all access control is done from a single configuration file that defines the whole security policy of a machine. Thus, there is only one ACL relating all subjects and objects in the machine.
based on several data contained in its credentials: domain and place of origin, class which implements the agent and user responsible for the agent. Before migration, these information, the initial state of agent and its code are digitally signed by the user that creates the agent. When an agent arrives at a remote site, its credentials are verified with regard to authenticity by checking the signature of the agent’s owner. The Ajanta platform uses a challenge/response authentication protocol with random nonce generation to prevent replay attacks, based on the signature of the agent’s owner.

In comparison to the static model in Java 2 and to the platforms discussed above, our scheme has the advantage of decoupling privilege attributes (credentials) from control attributes (policies), its use of some Java security features notwithstanding. This means that, although a policy configuration file still needs to be statically defined, the proposed mechanisms add the flexibility offered by SPKI certificates to domain generation. That is, domains are dynamically defined when an agent aggregates to its credentials the delegated certificates received during its itinerary.

Besides, in the agent authentication process described in section 5, the information used to determine an agent’s set of access rights is based not only on the identity of the agent’s owner, but also on the public keys of the owner and of the visited platforms, which avoids global name resolutions in large-scale systems.

Two related works propose the use of SPKI/SDSI certificates to improve access control in Java 2. The first, developed by Nikander and Partnen [19], uses SPKI authorization certificates to delegate Java permissions that directly describe possible permissions associated to a protection domain. In this work, the authorization tag of the SPKI certificate was extended to express Java permissions. This solution has the disadvantage of using modified SPKI certificates. The second work [20] proposes two improvements to access control in Java 2: association of access control information to each mobile code segment as attributes, and the introduction of intermediate elements in the access control scheme for assisting the configuration of access control attributes of incoming mobile programs and of access control information located in the runtime environments. The SPKI/SDSI group mechanism is implemented through name certificates and makes these improvements possible. Molva and Roudier’s work [20] does not provide details on how their proposal can be implemented nor on how to combine it to current Java 2 security model.

Both works discussed above do not deal with mutual authentication between source and destination platforms nor analyze the history of the visited platforms to establish trust on mobile code. Only the first proposal has flexibility characteristics similar to the ones proposed in the present work, in which the domains are formed according to the certificates delegated to an agent in its creation and throughout its itinerary. Nikander and Partnen propose that the search for forming new chains should be responsibility of the server. However, as mentioned before, this deviates from the SPKI/SDSI model.
8 Concluding Remarks

The acceptance of the mobile agent paradigm for developing distributed applications is still hampered by security issues. Limitations of the techniques mentioned in sections 2 and 7, provide evidence that they still do not present satisfactory results for providing security to mobile agent platforms. There are even more limitations when applications in large-scale systems are taken into consideration. In this context, naming and security solutions must always be flexible, scalable and open, allowing the evolution of such systems.

Our security scheme was motivated by perception of these limitations and a concern with the aspects of security specific to applications in large-scale systems. Its purpose is to prevent mobile agent attacks against platforms, defining a procedure that employs prevention and detection techniques: mutual authentication of platforms, authentication of multi-hop agents and generation of the set of permissions for an agent. This scheme is based on decentralized authorization and authentication control that is suitable for large-scale systems due to its use of SPKI/SDSI authorization certificates. The mechanism of SPKI authorization certificate delegation allows a separation between an agent’s privilege attributes and the control attributes (policies). The resulting scheme for generation of protection domains is more flexible than those of the related works.

The work described in this paper, although not fully implemented yet, already presents satisfactory results. As soon as it is concluded, its performance will be properly measured and evaluated. On the other hand, security is usually hard to prove and/or measure, due both to the complexity of systems and to limitations of the techniques available for the task. Security is thus much more amenable to qualitative evaluations than to quantitative ones. Therefore, in order to evaluate the security of our prototype, we are using testing methodologies prescribed by the Common Criteria, an ISO standard for qualitative security evaluation [21].

Integration and adaptation of this prototype to distributed Internet-based applications are also being done in order to demonstrate its usefulness [18]. Considering the protection of platforms from agents and of the communication channel, the proposed security scheme effectively mitigates the perceived security threats. Further work is still needed to propose techniques for protection of mobile agents against malicious agent platforms.

Acknowledgments

The authors thank the “IFM (Instituto Fábrica do Milênio)” and “Chains of Trust” project (CNPq 552175/01-3) members for their contributions. The first and second authors are supported by CNPq (Brazil). The third author is supported by CAPES (Brazil).

References

17. OMG: The common object request broker architecture v2.6. OMG Document 01-12-30 (2001)
Farm: A Scalable Environment for Multi-agent Development and Evaluation*

Bryan Horling, Roger Mailler, and Victor Lesser

Multi-Agent Systems Lab
University of Massachusetts
Amherst, MA 01003-9264
{bhorling,mailler,lesser}@cs.umass.edu

Abstract. In this paper we introduce Farm, a distributed simulation environment for simulating large-scale multi-agent systems. Farm uses a component-based architecture, allowing the simulation to be easily modified and augmented, as well as distributed to spread the computational load and improve running time. Technical details of Farm’s architecture are described, along with discussion of the rationale behind this design. Performance graphs are provided, along with a brief discussion of the environments currently being modeled with Farm.

1 Introduction

A tension exists in simulation frameworks which trades off the inherent richness of the provided environment, and the flexibility and ease with which that same environment can be used to analyze a particular aspect of a larger solution. On one hand, robust simulation environments can offer many enabling technologies that both increase the fidelity of the simulation, and provide a range of services that the participants may use to their advantage. These same features, however, can be an obstacle if the goal is to evaluate a particular technology in the absence of complicating factors.

Our prior work in the area of multi-agent simulation environments [14] resides in the former category; it provides a wide range of services in an attempt to create a realistic environment in which agents can perform and be evaluated. While using this approach is an important step in agent development, our experience has shown that it may also be helpful to extract key technologies from such an environment, and rigorously test them under conditions that have fewer distractions. For example, we will discuss a resource allocation technique that was more easily tested without many of the unrelated domain

* Effort sponsored in part by the Defense Advanced Research Projects Agency (DARPA) and Air Force Research Laboratory Air Force Materiel Command, USAF, under agreements number F30602-99-2-0525 and DOD DABT63-99-1-0004. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. This material is also based upon work supported by the National Science Foundation under Grant No. IIS-9812755. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA), Air Force Research Laboratory or the U.S. Government.

© Springer-Verlag Berlin Heidelberg 2004
complexities. Performing these more focused tests has the dual advantages of reducing possible artifacts from unrelated events, and improving the time needed for analysis by reducing the simulation overhead.

Our recent work addressing negotiation-based resource allocation [9] is a good example of this tension. The full-scale solution was developed by implementing fine-grained, sophisticated agents in JAF [14] using a detailed domain-specific simulation tool called Radsim [8]. Test scenarios were quite realistic, where agents were required to manage all aspects of a tracking multiple targets using a distributed network of sensors. This necessitated solutions for a range of issues, such as organizational design, dealing with noisy or uncertain data, managing agent loads, handling unreliable communication, disambiguating targets, etc [8]. Although each of these are important in their own right, and some have important effects on negotiation, many are orthogonal to the original resource allocation problem. We found that operating under such conditions not only distracted from this original goal, but also failed to illuminate potential flaws in the negotiation scheme. Negotiation errors were sometimes mis-attributed to related subsystems and we were unable to scale the collection of fine-grained agents using a reasonable number of processors.

In this paper we will present Farm, a distributed simulation environment designed to facilitate the analysis of the quantitative aspects of large-scale, real-time multi-agent systems. In particular, Farm provides essential base functionality needed to drive a multi-agent system, in such a way that elements such as the scalability, real-time convergence rate and dynamics of a particular system can readily be evaluated and compared. Farm has, in some sense, taken a step back by moving to a lighter weight implementation to provide an environment where multi-agent subsystems may be quickly developed and evaluated. Similar to [2], we are also interested in scaling up the size of scenarios, while reducing the time needed to run time.

Farm is a component-based, distributed simulation environment written in Java. Individual components have responsibility for particular encapsulated aspects of the simulation. For example, they may consist of agent clusters, visualization or analysis tools, environmental or scenario drivers, or provide some other utility or autonomous functionality. These components or agent clusters may be distributed across multiple servers to exploit parallelism, avoid memory bottlenecks, or use local resources. In addition, the set of components used in a particular scenario is not fixed - a limited set might be instantiated initially to reduce the simulation overhead, and components may also be dynamically added or removed at runtime as needed.

The agents operating in the system are the true subjects of the simulation. These are typically (but not necessarily) light-weight, partially autonomous entities that have a defined goal or role to fulfill within a larger domain context. Agents may be heterogeneous, either by instantiating the same type of agent with different parameters, or by using different classes of agent. Each exists as a thread, with its own local memory and control flow. To distribute the load incurred by the agent population, Farm organizes them into clusters, where each cluster exists under the control of a parent agent managing component that provides access to the rest of the simulation environment.

The agents themselves run in pseudo real-time, where individual agents are each allocated a specific amount of real CPU time in which to run. This aspect allows the
systems to exhibit a fair amount of temporal realism, where the efficiency of an agent’s activities can have quantifiable effects on performance in domains where the passage of time matters. For example, in the distributed sensor network domain we describe later, a sensor allocation scheme requires a certain amount of time to decide on an allocation. The amount of time available depends in part on the speed at which the target is moving. Because this time is accounted for in Farm, the target of the sensing has the chance to move away from the sensor while the allocation is being generated, producing lower utility despite having an otherwise valid allocation. This effect can be seen in Figure 1.

In that figure we compare the utility obtained by two allocation schemes as the target speeds are increased. The negotiate scheme requires time in which to compare allocations and communicate, while the random scheme simply picks sensors and ignores potential conflicts. Given enough time, the former produces a better result, while the latter gets a higher utility when time is scarce. A detailed discussion of these particular results is tangential here, we provide them as a motivating example of a situation where the temporal constraints of a situation are important, and can be captured by the environment provided by Farm. Communication actions are similarly modeled and monitored in such a way that message delivery times appropriate for the domain can be simulated.

Farm has been used to model five different domains, including a variety of agents implementing different types of solutions for these domains. Scenarios consisting of 5000 autonomous agents have been run using 10 desktop-class Linux boxes. These environments will be discussed in more detail later in this paper.

In the following section, we will provide a brief overview of the Farm simulator, followed by a discussion of how Farm relates to other MAS simulation environments. A more detailed look at Farm’s architecture is provided in section 3, along with a more in-depth examination of some of those features. We will conclude with examples of the environments that Farm has been used to create, and how Farm’s capabilities were used in their design.

2 Overview

As mentioned earlier, Farm is a distributed, component-based simulation environment. By distributed, we mean that discrete parts of the environment may reside on physically
Figure 2. An example of Farm’s component architecture.

Each part, or component, in this simulation environment is responsible for some aspect of the simulation. Figure 2 shows how a typical set of components in the simulation are related. The hub of activity is the Farm core, which serves as a connection point for components, and also drives the control flow in the system as a whole. The connected components then fall into two categories: those which directly manage the agents running in the system, and those which exist to support those agents or the scenario as a whole. These agent managers are called meta-agents, as each acts as an interface to and for a cluster of one or more agents. This organization is solely for efficiency purposes, it has no effect on the outcome of the simulation, and the individual agents do not know what meta-agent they are controlled by. Agents are implemented as threads, although this is only for performance purposes - from an agent’s perspective they are completely segregated, and are not aware of or share memory with other agents which happen to also be resident at the same meta-agent.

At runtime, agents are provided time in which to run, and other components (such as the drivers, analyses, and GUIs from Figure 2) are given the opportunity to perform, analyze, or modify the simulation at well-determined times. The run cycle is partitioned such that tasks like state maintenance or analysis may be performed without affecting the simulation results, even if they require indeterminate time. This is covered in more detail in section 3.1. Such tasks are facilitated by the ability to store and retrieve global information, which allows any given component to interact with a snapshot of the system’s current state. The design of this data storage is covered in section 3.2.

3 Architecture

As mentioned earlier, a simulation environment built using Farm is comprised of a number of components. Central to this arrangement is the Farm core, which handles com-
ponent, control and state management. The meta-agents, specialized components which manage clusters of actual agents, implement much of the architectural-level functionality supporting those agents. More generally, component plugins provide the remainder of the system’s non-agent capabilities, typically including both domain-independent and domain-specific elements which create, manage and analyze the environmental state. An arbitrary set of components may be used for a given simulation scenario. For example, one might choose to reduce simulation overhead by running without visualization, or with it to get a clearer picture of the system’s state. Multiple, different analysis components could be used to capture different aspects of the system. Different environmental drivers could be used, depending on what type of scenario is desired. Some additional discussion of the plugin types we have used can be found in section 4.

Whereas the types of the selected plugins are important in providing the capabilities and character of the simulation, it is just the total number of meta-agents which we are concerned about. The meta-agents are usually identical in functionality, and the size of the meta-agent set dictates how the load incurred by the agent population is distributed. Intuitively, in an environment with 100 agents, 5 meta-agents managing 20 agents each will run faster than a single meta-agent with all 100 agents. Experimental results looking at the effects of increasing the number of meta-agents can be seen in Figure 3, which shows its effect on total simulation time. The experimental setup consisted of a scenario containing 100 agents over a period of 60 seconds, using between one and five meta-agents. Recall that our goal is to provide each agent with an equal share of time, independent of what tasks the agents are actually performing. Ignoring the fact that an individual agent will in reality map to a single meta-agent, an optimal system’s duration would then exactly distribute the total runtime across all available meta-agents:

\[
\frac{\text{num}_\text{agents} \times \text{scenario_time}}{\min(\text{num}_\text{meta}_\text{agents}, \text{num}_\text{agents})}
\]

...or 100 minutes with a single meta-agent. This duration would decrease at a rate inversely proportional to the number of additional meta-agents until they reach a 1:1 ratio with the agent population. Farm closely models the behavior of an optimal system, with differences between the two largely attributable to Farm’s environmental modeling and component communication. The number of meta-agents can be increased arbitrarily, although it generally does not make sense to allocate more than one per processor.
Closely related to the total number of meta-agents is the load placed on each of them. Up to this point, we have assumed that agents allocated some duration of time will always require and use all that time. In practice, especially when agents can interact and may rely on one another for data, agents may have a considerable amount of idle time where they are awaiting a notification of one form or another. Farm allows agents to signal that they have no further actions to perform, so that they may preemptively release the time they have been allocated. In this way, they can reduce the actual runtime required, without changing the conceptual amount of time they have consumed in the simulation. If this capability is used, then the uniformity of agent load we have assumed so far may be incorrect, since some agents will be consuming different amounts of real processing time despite the fact that they all have identical simulated processing time. If there are different classes of agents, which may exhibit these different runtime characteristics, then this distribution should also be reflected in the allocation to avoid the need to estimate their actual computational requirements a priori. For example, if we have 5 meta-agents, 80 agents of type $x$ and 20 of type $y$, a good allocation would place $16x$ and $4y$ on each meta-agent because the load assigned to each meta-agent would be most evenly balanced. This scenario assumes no additional knowledge about the agents. If more details are available a more efficient allocation could be found, by reducing inter-meta-agent communication, for example. This will only affect the runtime characteristics of the simulation, and not affect the behavior of the agents themselves.

The environmental driver is responsible for this allocation; it is provided with the number of meta-agents, which it uses with its internal characterization of the desired scenario to decide upon an allocation of agents.

### 3.1 Control Flow

As the system starts up, the core acts as a registry for components, which contact the core as they are invoked. Components start by performing an initialization sequence, after which they wait for direction from the core.

Control in the simulation is concerned with the passage of time. Our ultimate goal in this is to ensure that each agent in the system is provided the same amount of physical CPU time, to evaluate how those agents would perform in a continuous-time environment. In a perfect simulation, all agents would be able to operate asynchronously in parallel. However, competition for the local processor by agents resident at the same meta-agent precludes this option if we wish to ensure fairness among them, and having one processor per agent is clearly infeasible for large numbers of agents. Thus, we approximate this behavior by sequentially assigning individual agents a slice of time in which to run. In between such opportunities an agent thread is paused, so the currently running agent has exclusive access to the CPU, as much as this is possible in a multi-tasking system. The simulation thus approximates how the agent population as a whole would act under real-time conditions, by breaking the timeline into a series of slices and providing each agent the appropriate amount of time to run for each slice.

This process is separated into two different components of the simulator: the core and meta-agents. The core starts a pulse by notifying all meta-agents in parallel that their agents may run for some duration of time. Each meta-agent then sequentially wakes their local agents, and indicates the specified amount of time which they have available.
to them. We refer to this process as the agents receiving a *pulse* of time in which to run. Ideally, the meta-agent would externally recognize when this time had expired, and halt the agent’s thread preemptively. However, this technique can lead to deadlocks and other undesirable, nondeterministic behaviors that affect the real-time performance of other agents in the simulation, especially if the agent is interacting with a remote third party at the time of the preemption. To address this, agents are provided with a simple time-checking function which should be called periodically, that internally tracks how much time has been utilized. When their allotment has elapsed, this function blocks and notifies the meta-agent of completion. At the beginning the next pulse, the thread is restarted from this same location, so only a minor intrusion into the agent’s code is required.

Just before and after this pulse is sent to the meta-agents, all components in the simulation are allowed an indeterminate amount of time in which to run. For example, before the agents are pulsed, a driver might update environmental data (e.g. a moving target’s position). After a pulse, analysis tools might take time to update statistics or visualizations. Because the agents are halted during this time, and the duration of these activities will not affect the results of the simulation, the designer is free to perform any number of activities during this period (at the expense, of course, of possibly increasing how long the simulation takes to run). The process of providing time before, during, and after each pulse is repeated by the simulation core until the simulation is completed.

Because of this style of control flow, interactions do not take place between agents within a single time pulse - the effects of one agent will not be observable by another until that pulse has ended. This can lead to a certain amount of data, communication or behavioral incoherence in the system, the duration of which is bounded by the length of the time slice. For example, messages cannot be delivered faster than the pulse duration. In another situation, the effect of one agent’s action will not be observable by another immediately, but must wait until the following pulse when the result of that action can be shared. Mechanisms for addressing this issue are covered in section 3.5.

The pseudo real-time nature of this control used by Farm implies a certain amount of non-determinism to otherwise identical scenarios, as external events may effect the actual amount of processing time an agent receives during a window of real time. If determinism is required, Farm also supports a fixed notion of pulse time, which tracks the agent’s execution progress, rather than just elapsed time. For example, instead of allocating 100ms per pulse, agents could be allocated a single pass through their main event loop. Thus, for each pulse, each agent’s *pulse* method would be called once, and (assuming the agents themselves are deterministic) the scenario as a whole will be deterministic because the same sequence of activities will be performed for each run. This allows repeatability, but prevents one from drawing strong conclusions about how the agents behave in real time.

### 3.2 Data Flow

The Farm provides the components in the simulation with a data storage facility with characteristics similar to distributed shared memory. This is not inter-agent shared memory (agents are still assumed to interact via messaging), but instead provides an indirect means of interaction between the simulation components and a way to deliver environ-
mental information to the agents. Using this scheme, components may store and retrieve data (properties) from a functionally common repository, enabling the data produced by one part of the simulation to be used by another. For example, an environmental driver might be responsible for updating the Target1:Location property. Agents needing to know that target’s location can then simply access this property. Similarly, each agent could store some notion of it’s current state in, for instance, the property Agent1:State. An analysis component could then find all properties *:State to capture a snapshot of all the agents in the system.

Distributed data storage is accomplished in Farm through the use of a token system. Each globally-accessible property is associated with a token, which may be resident in the simulator core or at any one of the plugins. The owner of the token for a particular property is responsible for its storage; all reads and writes to that value are performed by or through it. The Farm core itself is responsible for keeping track of who owns the token for each property, somewhat like a specialized directory service. When a property is to be read for the first time by an entity, it optimistically assumes that it is stored at the core, and makes a request to the simulator for it. If it is being stored there, the property’s data will be delivered and the process continued uninterrupted. If a different plugin is the owner of that property, the core instead provides the requester with the name of that plugin. This can be used to contact the appropriate plugin and retrieve the data. This property-plugin mapping is then cached so future requests can be made directly to the owner plugin. Because plugins may leave, have their tokens removed, or otherwise lose control of a property, such future requests may fail. In this case the requester will again revert to the default assumption that the core itself is storing the property, since the core will either return the data or redirect the reader as needed.

Property writes occur in a similar manner. If the writing entity has a cached property-plugin mapping, then it will contact the appropriate plugin with the new data to be written. If no such mapping exists, or if the property is controlled by the core, then the simulator itself is contacted. As with reading, the simulator may store the data itself, or redirect the writer to the appropriate owner plugin. Because property owners may change over time, writers will again fall back to the default assumption that the property may be controlled by the core if the local knowledge is out of date. We will assume from here on that the plugins fulfill all of the storage responsibilities; simulation core storage is generally only used as a fail-safe mechanism in case a plugin fails or is otherwise unavailable.

An additional mechanism also exists which allows plugins to be automatically updated with a property’s data when it is updated. A list of recipients is attached to the owner’s token, so when the property is changed the new value can be automatically pushed to each member of the list. This pushed data is flagged as being cached, so the recipient knows it can safely read from the data, but writes must still be propagated back to the owner. The owner is responsible for ensuring that this remotely cached data is kept up to date.

Farm also Java’s RMI and serialization services to support mobile code, for cases where data retrieval bottlenecks are otherwise unavoidable. With this technique, instead of remotely retrieving and locally processing a potentially large amount of data, a specialized function is delivered to and run by directly by property owners. This function can be written to use only local data, with the intention of returning a more concise view
to the originator. This is particularly useful for analysis components, which frequently need to access data that scales in number with the agent population. The dominating costs of this technique scale with the number of plugins, resulting in substantial savings in bandwidth and time.

### 3.3 Communication

To improve scalability, communication takes place entirely outside of the core. Instead, communication occurs between meta-agents, and individual agents send and receive messages via their managing meta-agent. When a new meta-agent registers with the core, all existing meta-agents are told about the addition, and the new meta-agent is given a list of all other members - thus a fully-connected graph of meta-agents is maintained. When an agent sends a message, it is added to a per-agent outgoing queue. The meta-agent selects ready messages from these queues and checks its address table to determine the recipient’s owning meta-agent. If the meta-agent is found, the message is delivered. If it is not found, it uses the list of known meta-agents to find the appropriate one, and that mapping is then recorded in the address table. Thus each meta-agent will learn a mapping for only necessary destination agents. As messages are received by a meta-agent, they are added to a per-agent incoming message queue, which is polled by the agent as necessary.

We wish to have a relatively realistic network model, so care is taken when sending messages. A potential race condition also exists for message delivery, as one agent’s message may reach another agent before it has technically been sent in the global time line. As messages are added to an agent’s outgoing queue, they are marked with a delivery time. The delivery time of a message will be that of the prior message in the queue, plus a bounded random transit duration which can be weighted by the length of the message. A message loss rate probability may also be set. At the end of a pulse, each meta-agent searches the outgoing message queues of its local agents, and sends messages if permitted by the assigned delivery times. These messages are queued for delivery at the destination meta-agent. At the beginning of the next pulse, those received messages are delivered to the appropriate incoming queue for each agent. The agent is then responsible for monitoring its queue and handling new messages. We are investigating other potential communication paradigms, such as a defined routing network or distance-limited broadcast.

While this decentralized communication mechanism scales very well, it prevents other components from directly observing or analyzing message traffic. Gross statistics, such as total incoming and outgoing messages, are currently computed and stored as global properties by individual meta-agents. Other statistics can be computed in a similar manner to compensate for this design decision.

The exact messaging protocol is left intentionally unspecified and abstract. Agents simply send `Message` objects, which can be extended as needed. The destination agent then receives `Message` objects in its incoming queue. Parsing of the object is performed automatically.

### 3.4 Scalability

Some discussion of the scalability of Farm has been mentioned earlier. The component architecture of Farm, and specifically its ability to segregate the agent population into
groups under the control of distributed meta-agents, leads the environment to large scale scenarios. Because the agents effectively run in parallel because of this distribution, the primary constraint is having available computing power to run such simulations in a reasonable amount of time.

Figure 4 shows a sample of Farm’s scalability characteristics, from the results of a series of repeated trials with a scenario length of 60 seconds. The number of meta-agents was fixed at five, and the number of agents gradually increased from 100 to 1000, with a 1:4 ratio of targets to sensors. The distributed resource allocation domain from section 4.1 was used, because the movement of targets provides a need for continual re-evaluation and communication over areas of contention. The initial results seen in this graph are promising.

In comparing the 100 agent point from Figure 4 with the 5 meta-agent point in 3 (which also used 100 agents), one might also note a significant difference in simulation duration. For example, the 100 agent case here took only 6 minutes, as compared to 124 minutes previously. Both experiments used the same machines, domain and agent population, but the trials from Figure 4 allowed the agents to signal their meta-agent if they have no additional work to do, using the technique mentioned earlier in section 3. This allows the agent’s pulse cycle to be ended prematurely, with potentially large savings in actual running time without loss of precision in the simulation results. In our scenario, if there are more agents in an environment of constant size, there is a higher probability that additional computation will be needed to resolve the correspondingly larger number of conflicts. This is seen in non-linearity of the data in Figure 4, where disproportionately more time is used in larger populations. In this way, the system avoids expending effort simulating agents’ “idle” time, which gives Farm some of the benefit that a strictly event-based simulation environment would posses.

Perhaps more interesting than “how large can it get?” is the question “what prevents it from getting large?” No design is perfect, and parts of Farm’s architecture can inhibit scale in order to permit other features. Typically, the most constraining feature is the remote access of data, as outlined in section 3.2. This is necessary to facilitate state analysis, but excessive usage can accumulate a large time penalty. In general, such scenarios will just take longer to process than if the data storage were completely distributed. If an appropriate allocation strategy is used, then in many cases data caching and mobile code can ensure agents are not unduly penalized for the time required to update remote data.
Another constraint, related to data flow, is environmental maintenance. The task of creating and maintaining the simulation environment (e.g. placing sensors, moving targets, etc.) is typically the responsibility of a single component. Like any other, this component may be distributed for load balancing purposes, but it is still a single process limited to the resources present at its local processor. Like the agents, it also accesses state data, but since it has the responsibility of maintaining the entire state, the potential burden is much more concentrated. Extremely large, complex or dynamic environments might therefore benefit from separating this responsibility into separate components, much as the agents themselves are separated. Thus, one might have a target component, a sensor component, and the like, each with a specific, tractable responsibility.

3.5 Coherency

Whenever an environment is distributed, the problem of coherency arises because entities on one processor may have data inconsistent with that on another. One must try to make sure that interactions between processors are as faithfully represented as those occurring on the same processor.

The data consistency problem in Farm manifests itself in the time between when one agent changes a value to when that change can be observed by another. In between those events, the system can lose some measure of coherence. Because Farm is attempting to replicate the behavior of a centralized simulation, data storage should exhibit strong coherence, and components always use the most up-to-date property when possible. One could envision a system where a weaker form of coherence might be acceptable to some components (e.g. visualization) in order to reduce overheads. In this case, techniques used to minimize the cost associated with maintaining such a heterogeneous population could be employed, as shown in [12]. However, due to the relatively small population of plugins in a typical scenario, the costs associated with maintaining such special cases may outweigh the potential benefits. We have previously outlined Farm’s data dissemination technique in section 3.2. In the absence of cached information, coherence is maintained by the single owner for each property, which handles all read and write accesses for the property. These accesses are serialized by the owner, ensuring that consistency is maintained. In the case where data has been pushed to and cached by remote components, the owner is also responsible for immediately updating those components with the new information. Owner coherence is maintained by locking the property for the duration of the update, while remote coherence is limited by the network and processing delay incurred by the update.

Communication coherency is also important. Farm must ensure that a message is delivered when appropriate, and from the recipient’s perspective, not before it was actually sent. As outlined in section 3.3, messages from the agents are queued for delivery during the pulse, and only sent after the pulse has completed. The receiving meta-agent queues incoming messages, which are delivered to their final recipient when the specified delivery time has been reached.

A more insidious form of inconsistency occurs when the meta-agents are distributed across a heterogeneous set of machines. Because the agent’s computational effort is measured in seconds, one group of agents may effectively be allocated more time simply because the processor they happen to reside on can perform more computations in the
same amount of time. A few strategies can be employed to compensate for this problem. One could compute a processor-to-real time ratio for all machines in the pool, and use that to scale the actual time allocated by individual meta-agents. In Farm, this is accomplished by first determining a baseline performance standard, either by prespecifying it or by having the simulation core to dynamically compute one from the computer it resides on. This baseline value is then provided to the meta-agents, who dynamically determine their individual performance metrics by computing a common benchmark. We use the Linpack benchmark [1] to produce these performance measures. The ratio of the baseline value to a meta-agent’s individual performance metric is used to weight the execution time it allocates to the agents under its control.

Another strategy to address the issue of heterogeneity is to statistically remove the problem through repeated trials where the agent population is shuffled between hosts. A third option (clearly requiring less effort) is to simply ensure your server pool is sufficiently homogeneous, or accept the performance differences as a byproduct of working in a realistic environment. For the results presented in this paper, the experiments were performed using a group of similarly configured workstations.

4 Environments

Several computational environments have been implemented using Farm, each taking about two days to implement the environment itself, and the agents taking from a day to a week depending on their complexity and the availability of source code. Each environment generally consists of a driver, which instantiates and maintains the environment, and analysis components, which generates statistics at runtime, and a set of one or more types of agents. In addition, several generic components have been developed which may be used across all environments. These include a graphing component, property log, time driver, and simple statistical analysis (running averages and standard deviations).

4.1 Distributed Resource Allocation

The distributed resource allocation environment is an abstraction of the distributed sensor network (DSN) problem [8]. A complete solution would reason about a range of issues, from role assignment to low level task scheduling to multi-target data association. The underlying problem, however, is much more straightforward. The environment consists of a number of sensors and mobile targets, and the high level objective is to use the sensors to track them. Each sensor has limitations on its range and usage. This then reduces to a resource allocation problem, where the attention of the sensors must be allocated so that all the targets are tracked.

Several plugins were created for this domain. The first was a driver, which is responsible for assigning sensor and target agents to meta-agents, and for creating and maintaining the sensor environment. This includes determining locations for the simulated sensor nodes, updating target locations as they move, and maintaining sensor-target observability lists. These lists were stored as properties, so that they can easily be accessed by the individual agents. This and other data was also used by an analysis plugin, which determined the possible and current utility produced over time, tracked message totals, and sensor allocation conflicts, among other things. Much of the data it uses are
produced by individual agents, which are then stored as global properties. One can see that as the system scales, this data transfer can become significant, which motivates the property storage discussion presented earlier in section 3.2. Farm’s generic graphing plugin, seen at the bottom of Figure 5, was used to visualize some of this data. The analysis component works by aggregating the data and producing some sort of summary statistic at each time pulse. This statistic is also stored as a property, the name of which can be provided to the graphing plugin at launch time so it can find and graph the values over time. Another domain-specific visualization component, shown at the top of Figure 5, displays the current sensor field, along with target allocations, conflicts and negotiation patterns. This works in a manner similar to the analysis component, aggregating data produced by individual agents with location data from the driver to produce its image. The same data collection costs mentioned above also motivate the need for a component-based system such as Farm employs that allows one to run both with and without potentially expensive components like these visualizers.
Our comprehensive solution to this problem is implemented as a homogeneous collection of sophisticated JAF agents, which run in real time in both the Radsim simulator and hardware. In the Farm environment there are two types of simpler agents: sensor agents, each of which controls a single sensor, and tracking agents, each of which is responsible for tracking a single target. The driver provides the track managers with a list of candidate sensors, i.e. those which are in range of its target, and the track manager must determine which sensors it wants to use. The track managers must then coordinate to resolve conflicts so all targets are tracked. The SPAM negotiation protocol [9] was implemented to solve this problem.

This domain was the incentive behind Farm’s creation, and has shown itself to be particularly useful in debugging and evaluating SPAM. Because Farm scales to much greater numbers, and also eliminates most of the complicating, but ultimately tangential factors (relative to resource allocation), development and evaluation of the protocol was much easier in this environment than in the original detailed simulator. We were also able to directly use almost all the code from the original JAF-based implementation, so improvements made in the Farm environment were easily mapped back the more realistic Radsim and hardware environments.

### 4.2 Graph Coloring

The well-known graph coloring domain was implemented as a means of both testing the generality of SPAM, and also to compare its performance against reference protocols known to work on graph coloring. The driver in this domain was simpler than for DSN, as it only needs to create the nodes and edges of the graph. A domain visualizer was written to help observe system behavior. We use the layout procedure described in [15] to produce satisfiable graphs of arbitrary size using a defined number of colors. Currently the resulting graph is static, although we intend to add an additional dynamic component to it in the future. A separate analysis component evaluates the possible and actual number of coloring constraints, which is then visualized using the graphing component. Three agents have been implemented in this domain, using protocols derived from descriptions in [15].

### 4.3 Learning Evaluation

The learning domain we developed is somewhat different, in that was not intended to produce an environment where a number of agents work together addressing a common environment. Instead, a number of individual agents learn in parallel on separate problems. Specifically, we have implemented a simple $n$-armed bandit problem [13], along with several agent types instantiating different reinforcement learning techniques. The purpose of this was to create an environment where a number of tests can be run concurrently, to provide more immediate, comparative feedback as visualized by Farm’s graphing plugin in Figure 6. In this figure we compare how the percentage of agents performing the optimal action changes over time between two different techniques, where each measurement for each technique is the average over the behavior of 500 concurrently running agents.
The driver for this domain was responsible for determining what types of learning agents to create. As with the DSN domain, this is a potentially heterogeneous collection of agents, so allocation was performed by distributing a representative sample of this collection to each meta-agent. The learning domain also employed a more complex analysis component, which used mobile code to first generate aggregate statistics at each component, and only transfer a summary back to the analysis component. These summary statistics were then combined to produce the global statistic. This produced a large savings in time and communication, at the expense of a more complex implementation.

This particular domain is also differentiated by its use of the deterministic notion of pulse time. Agents are controlled so that instead of running for a certain number of milliseconds, they are allowed only a fixed number of passes (1) through their control loop per pulse. This allows us to ignore the computational costs of the learning techniques (which might vary by implementation), and instead focus on their theoretical convergence rates. This control style may be turned off to analyze real-time performance.

5 Related Work

Attempts have been made [10,6] to define the set of features and characteristics that a multi-agent simulator should possess. The topics described in these efforts are important, in that they can help guide designers towards a comprehensive, robust solution. Farm, however, is not intended to provide a complete simulation solution - instead it tries to provide a relatively simple environment where agents possess only the most germane functionality. Much of the complexity of a real environment can be either abstracted away, or approximated through the use of black-box style components which provide agents with necessary information, when the actual process of obtaining that information is unimportant. Thus, much of the underlying modeling structures which make other simulators unique is absent in Farm.

5.1 MACE3J

MACE3J [4], like Farm, is primarily intended to simulate large numbers of large-grained agents. It includes mechanisms for different styles of event and messaging control, data collection, and a lightweight agent definition model. It is also scalable, but does so under a multiprocessor-based scheme, taking advantage of the capabilities inherent in lower
level system software to manage many of the inter-processes and inter-thread issues which arise in simulation. While this method is undoubtedly more efficient than the distributed approach we have selected, it also requires additional physical overhead in the form of an actual multi-processor machine, or a cluster of machines tied together with appropriate software. In this sense, Farm is more closely related to the original MACE [3], which also employed a distributed architecture.

Farm places more emphasis on the real-time aspects of agent behavior, as progress in a scenario is driven by the passage of time, not events or messages, and the effectiveness of an agent is affected by the duration of its computations. In contrast, MACE3J’s event driven control with synchronization points allows for more deterministic results. In other ways the two environments are similar: Farm also supports repeatability, varied communication models, and data collection and display tools.

5.2 Swarm
Like Farm, the Swarm [11] simulation environment is a modular domain-independent framework. It offers the ability to probe agents’ state and beliefs, and graphically display them, similar to the logging and graphing tools provided with Farm. Fundamentally, the two differ in their representation of time. Swarm uses a discrete event system, where a point in time is not reached or seen until some event has been scheduled to take place. Farm uses a real-time approach where time passes regardless of what events are taking place. Both techniques are valid, but serve different purposes. In addition, Swarm agents have a different character to them, as they are generally modeled as a set of rules, or responses to stimuli. Conversely, Farm agents are built more like a conventional program, where classes and routines are built to exhibit agent behavior.

5.3 CoABS Grid
The CoABS Grid (CG) [7] is a middleware system designed to integrate a collection of architecturally heterogenous agents into a single consistent environment. It has many of the characteristics of a simulation environment, although it is more robust in that the environment and the effects of agent activities need not necessarily be simulated. CG has some features that directly provide domain services to the agents, such as capability registration and search, secure communications, and directory services, which in a Farm simulation would traditionally be considered part of the agent solution. However, in the spirit of avoiding unnecessary complexity that we have espoused, one could envision, for example, a Farm directory service plugin in much the same way drivers are used to relieve the agents of other burdens. Like Farm, CG strives to be a scalable solution, has support for optional visualization and logging tools, mobile code, and fully distributed point-to-point communication. Farm differs from CG in its tighter control of time, in that agents are allocated a particular amount of execution time by the Farm core. Because CoABS agents may be legacy systems or non-native wrapped agents this level of timing control and accountability is difficult to achieve in that framework.

5.4 MASS
Our earlier simulation environment, the Multi-Agent System Simulator [14], is quite different than Farm. It provides a richer, quantitative view of the world, where agent
activities and their interactions are modeled using a hierarchical task decomposition language, consumable and non-consumable resources have constraints which can affect behavior, and an agent’s beliefs may differ from objective fact. As with the earlier example, agents are built using JAF, which itself has a fair amount of complexity. All of these features are desirable for evaluating sophisticated agents in context, but at the same time they can be distracting when only a subset of behaviors need analysis. In addition, the environmental models and communication mechanisms are centralized, and the agents, while distributed, run as separate processes, so the environment as a whole does not scale well past 40 agents or so. The DECAF [5] agent framework also has a similar character and purpose to JAF/MASS, although it does not have a centralized simulation environment, and it offers built-in brokering and name services which JAF lacks. In most other respects, DECAF compares to Farm in much the same way as JAF.

5.5 VMT

The Vehicle Monitoring Testbed (VMT) was one of the first distributed simulation environments designed to support distributed intelligent processing [2]. It is interesting to note that the same capabilities which we currently address in Farm, such as the desire to scale, overcoming resource bottlenecks, and facilitating distributed control, were the same motivating issues in a system designed 20 years ago. Although the VMT environment has a more specialized purpose, it still shares several features with Farm, including a similar messaging scheme, a notion of global time and a synchronization mechanism ensuring distributed processes receive equal amounts of time.

6 Summary

Farm is a multi-agent simulation environment designed to handle large scale simulations using custom analysis, visualization and environments, while tracking agent activity in simulated real time. The main simulation entity acts as a hub, by accepting and managing connections from distributed plugins, providing execution prompts to those plugins, and maintaining a globally accessible data repository. Agents in the system are implemented as threads, but are autonomous in character, require communication to interact, and do not share memory. These agents are organized in groups on distributed processors, where their real-time CPU usage is monitored and rationed in accordance with the simulated environment’s design. By distributing both the agents and analysis tools, Farm is able to exploit available computing power to handle very large environments, while retaining the ability to effectively model real world performance.

Issues relating to scale and coherency are closely tied to the distributed nature of the system. On one hand, poor data and computational distribution can lead to communication and processing bottlenecks. On the other, because agents are distributed across processors, care must be taken to ensure temporal, data and communication consistency. Different strategies for managing these issues were covered.

The environment so far has been used to create scenarios containing more than 5000 individual agents. Several domains have also been implemented, including distributed sensor network and graph coloring testbeds.
References


Role-Based Approaches for Engineering Interactions in Large-Scale Multi-agent Systems

Giacomo Cabri¹, Luca Ferrari¹, and Franco Zambonelli²

¹ Dipartimento di Ingegneria dell’Informazione – Università di Modena e Reggio Emilia
41100 Modena, Italy
{cabri.giacomo, ferrari.luca}@unimo.it
² Dipartimento di Scienze e Metodi dell’Ingegneria – Università di Modena e Reggio Emilia
42100 Reggio Emilia, Italy
zambonelli.franco@unimo.it

Abstract. This chapter discusses how the concept of role can be exploited in engineering large-scale multi-agent systems, in order to support the development of the applications and to simplify the related tasks. A role can be considered as a stereotype of behavior common to different classes of agents. Role-based approaches can give several advantages, the most important of which is the separation of concerns between different issues (for instance, algorithmic ones and interaction-related ones), which is very useful to simplify the development of large-scale agent-based applications. A survey of different approaches shows the advantages and the peculiarities of introducing roles in agent-based application development. Moreover, we present in detail the BRAIN framework, developed at the University of Modena and Reggio Emilia, which is an approach based on roles to support agent developers during their work.

1 Introduction

Agent sociality leads to autonomy in interactions, allowing scalable decompositions of large-scale applications in terms of decentralized multi-agent organizations [15], and enabling interaction among agents not only belonging to the same application but even to different ones, as happens in the people real world. Therefore, interactions are an important issue to be faced in agent-based applications, both whether they occur between cooperating agents of the same application and between competitive agents belonging to different applications. Sociality, proactiveness (i.e., the capability of carrying out their goals) and reactivity (i.e., the capability of reacting to environment changes) are considered the main features of the agents [22], and in our opinion must all be taken into account in developing interactions between agents. The fact that agents can move from host to host adds further flexibility, but also introduces peculiar issues in interactions [13]. In the following we consider two scenarios, ubiquitous computing and the Web, to discuss about interaction requirements in large-scale distributed environments.

An approach that seems to be useful in this field is the exploitation of the abstraction concept of roles. Roles represent a concept that have been exploited in several software engineering fields, such as UML [37] or RBAC [36], and this emphasizes how roles can be useful in engineering interactions; of course, they must be adapted to
be applied to the agent scenarios. A role is a stereotype of behavior common to different classes of agents [17, 26] and can generally be considered as a set of behavior, capabilities and knowledge that agents can exploit to perform their task(s). There are different advantages in introducing roles in interactions and, consequently, in exploiting derived infrastructures. The most important one for large-scale development is that the use of roles enables a separation of concerns between the algorithmic issues and the interaction ones. In the literature we can find different role-based approaches addressing different phases of the multi-agent system development. This chapter presents a survey of different approaches and point out their main limitations; in our opinion, the most important limit is that the presented approaches address only single aspects of the agent-based software development (for instance, some propose a model, while other focus on implementation) and a more complete approach is missing.

After a survey of existing role-based approaches, we present in detail the BRAIN (Behavioral Roles for Agent INteractions) framework, developed at the University of Modena and Reggio Emilia [10]. BRAIN proposes an approach based on roles whose aim is to support the development of interactions in agent-based applications, overcoming the limitations of existing approaches. To this purposes it provides for (i) a simple yet general model of interactions based on roles, (ii) an XML-based notation to describe the roles, and (iii) implementations of interaction infrastructures based on the previous model and notation, which enables agents to assume roles and interact. In particular, BRAIN proposes a specific definition of role, which is not in contrast with the previously mentioned general definition, but it is quite simple and can be easily exploited in concrete implementations. In BRAIN, a role is defined as a set of capabilities and an expected behavior. The former is a set of actions that an agent playing such role can perform to achieve its task. The latter is a set of actions that an agent is expected to manage in order to “behave” as requested by the role it plays. Interactions among agents are then represented by couples action-event, which are dealt with by the underlying interaction infrastructure, which can enforce local policies and rules.

The structure of this chapter is the following. Section 2 proposes two large-scale application scenarios that propose challenging requirements that multi-agent application developers are going to meet. Section 3 reports on different role-based approaches that can be found in the literature. Section 4 describes the BRAIN framework. Section 5 shows the usefulness of roles, and of BRAIN in particular, by means of an application example. Finally, Section 6 concludes.

### 2 Background and Motivations

Two significant examples of complex software systems that introduce challenges in large-scale application development are pervasive computing and Web-based services. In this section we list the main requirements emerging in such scenarios and sketch how role exploitation can meet such requirements.

Pervasive (or ubiquitous) computing aims at assist users in carrying out their tasks by means of a computing infrastructure that is ubiquitous and invisible [39]. Such infrastructure is composed of several connected entities [4, 14], mainly sensors (which keep track of the evolution of the physical environment) but also devices that interact with people. Such a scenario is highly dynamic, both because the entities belonging to the same context may change very quickly and because there is a lot of
pieces of information that must be evaluated. The different entities are heterogeneous (i) as kind, ranging from the so-called smart dust (small networked sensors) [25] to powerful notebooks, (ii) as devices’ capabilities, and (iii) as running OSs and applications. Moreover, in such a world the failures are not exceptions, but normally occur, leading to unpredictability that must be faced by entities in an autonomous way. From our point of view, one important requirement of ubiquitous computing is spontaneous interoperation [28], i.e., the capability of different entities of interacting, without human intervention, to carry out a task. Very different from the current interaction among computers, it requires autonomy in discovering other entities, adaptability in talking, and dynamism in assuming appropriate roles [32].

The other challenging scenario is constituted by Web-based services. Internet and the Web have rapidly spread all over the world, and are clearly influencing the everyday life of millions of people. It is a scenario more mature and spread than pervasive computing, and the exploited technologies are quite settled. However, it is still evolving, shifting from a bare information container to a provider of several kinds of service. This does not calm the call for new methods and tools to face the application requirements introduced by such an evolution. The content of the Web is required to be more dynamic, to suit on demand services. The willing of exploiting the Internet everywhere has led to the need of facing the heterogeneity (in terms of capabilities, OSs and applications) of devices, such as cellular phone, PDA, and whatever the device technology will propose. Today’s and future Internet applications are component based, and must be open and adaptable. Also in this scenario, the interaction among different components is a relevant issue because have to take into consideration heterogeneity, dynamism, and unexpected situations.

Pervasive computing and Internet scenarios are converging, because they exhibit similar properties and both call for software development methodologies based on components, capable of dealing with the described characteristics in an autonomous way while carrying out tasks. A suitable recognized approach is the agent-oriented one, which is based on the definition of autonomous software entities that carry out tasks on behalf of users, reacting to the changes of the environment where they live [20, 43]. So the agent-based approaches seem to provide the adequate abstractions for building complex software systems [21], thanks to their autonomy, which help application developers in identifying responsibilities, and their adaptability, which enables them to face dynamic situations [27].

Moreover, as mentioned, in the two challenging application scenarios, interactions are an important issue to be faced. In the multi-agent field the problem of dealing with interactions has been more focused. In fact, one of the main features of agents is sociality, i.e., the capability of interacting each other; interacting system components can be thought as organizations of agents. Multi-agent system massively exploit the feature of sociality, since the main task is divided into smaller tasks, each one delegated to a single agent; agents belonging to the same application have to interact in order to carry out the task [15]. Moreover, scenarios such as pervasive computer and the Web-based services imply that not only agents of the same application interact in a cooperative way, but also agents of different applications may interact in a competitive way, for example to achieve resources. The feature of mobility, which allows agents to change their execution environment, adds great flexibility at both conceptual and implementation levels, but also introduces peculiar issues in interactions, such as localization, site and platform dependences, which must be taken into account [13].
We argue that an important aspect in the development of multi-agent applications is the separation of concerns between algorithmic issues and interaction issues. This facilitate developing applications because allows facing the two issues separately, leading to a more modular approach. This is one of the main aims of our approach. Besides this, the depicted scenarios point out the following non-functional requirements for the development of agent interactions in large-scale applications:

- **Generality.** Approaches should be quite general, and not related to specific situations, to be effectively adaptable and flexible.

- **Locality.** A trend that has recently appeared and seems to be accepted in the agent area is the adoption of locality in agent interactions, i.e. the environment is modeled as a multiplicity of local interaction contexts, representing the logical places where agent interaction activities occur. Due to its movements, depending on its current location, an agent is situated on a given interaction context and there will be enabled to access local resources and to interact with local executing agents [8].

- **Reusability.** Developers should not reinvent the wheel for any new application. A valid approach must enable an easy and wide reuse not only of code but also of solutions.

- **Agent oriented features.** In our opinion, the most important requirement is that interactions have to be modeled following an agent-oriented approach, i.e., all the peculiar features of agents must be taken into account.

- **Practical usability.** Besides formalisms for modeling interactions (such as UML-based approaches to AOSE [30, 34]), we argue that the agent interaction development must be supported and simplified in a concrete way.

To meet the above requirements, a clear and simple model must be adopted, and, on the base of it, tools and infrastructures must be built to support the developers of agent-based applications.

There have been different proposals in the area of agent interaction and coordination. They have concerned message passing adapted to agents, “meeting point” abstractions [40], event-channels [2], and tuple spaces [6]. However, these approaches to agent interactions suffer from being adaptations of older approaches traditionally applied in the distributed system area and do not take into account the peculiar features of agents (mainly proactiveness, reactivity and sociality), leading to fragmented approaches. Some efforts in this direction model interactions in terms of services and tasks [18], tailoring interactions on agent features. Moreover, traditional approaches often consider agents as bare objects; this implies that the different features of agents are managed in a not uniform way, leading to fragmented approaches. Finally, there is no approach that covers several phases of the software development, leading to fragmentation of the solutions.

The approaches that seem to overcome the mentioned limitations are the one based on the concept of role. As mentioned above, a role is a stereotype of behavior common to different classes of agents [17] and can generally be considered as a set of behavior, capabilities and knowledge that agents can exploits to perform their task(s). There are different, well-recognized advantages in modeling interactions by roles and, consequently, in exploiting derived infrastructures. First, it enables a separation of concerns between the algorithmic issues and the interaction issues in developing agent-based applications [10]. Second, being a high-level concept, roles allow independence of specific situations, and promote quite general approaches. Third, roles
can be developed according to local needs and laws, promoting locality in the development of large-scale distributed applications. Fourth, roles permit the reuse of solutions and experiences; in fact, roles are related to an application scenario, and designers can exploit roles previously defined for similar applications; roles can also be seen as a sort of design patterns [1, 31]: a set of related roles along with the definition of the way they interact can be considered as a solution to a well-defined problem, and reused in different similar situations. The last two requirements previously mentioned (the agent oriented features and the concrete usability) do not derive from roles, but must be met by specific approaches.

3 A Survey of Existing Approaches

In this section we propose a survey on role-based approaches in the area of agent-based application development. The concept of role is not exploited at same way in all approaches, and usually it is used in only one phase of the application development. For instance, the first two approaches uses roles at the model level, others at the implementation level, and further ones in a mixed way.

Gaia [41] is a methodology for agent-oriented analysis and design. Its aim is modeling multi-agent systems as organizations where different roles interact. In Gaia, roles are used only in the analysis phase and are defined by four attributes: responsibilities, permissions, activities, and protocols. Gaia exploits a formal notation (based on the FUSION notation [11]) to express permissions of a role. Even if it is not a notation to completely describe roles, it can be helpful because enables the description of roles in terms of what they can do and what they cannot. In addition, Gaia proposes also an interaction model that describes the dependencies and the relationships between the different roles of the system, and it is considered a key point for the functionalities of the system. In Gaia, the interaction model is a set of protocol definitions; each protocol definition is related to a kind of interaction between roles. A possible limitation of Gaia is that roles are considered only in the analysis phase.

AALAADIN [16] is a meta-model to define models of organizations. It is based on three core concepts: agent, role and group. The last is a set of agent aggregation, which can be considered atomic with regard to a task to accomplish or to dependencies between agents. The group concept can be useful when a large number of agents are involved, as may happen in large-scale applications: in fact, it enforce modularity and allows an easy decomposition of tasks, still keeping the single agents simple. In AALAADIN roles are tightly bound to the notion of agent, and this can become a drawback if developers’ aim is to describe roles independently of agents.

The ROPE project [3] recognizes the importance of defining roles as first-class entities, which can be assumed dynamically by agents. It proposes an integrate environment, called Role Oriented Programming Environment, which can be exploit to develop applications composed by several cooperating agents. Since this approach strictly focuses on collaboration, in our opinion, it lacks to address the interaction between competitive agents, while other approaches are more flexible and provide roles also for interactions between agents that do not belong to the same application; this is a relevant aspect in the design of applications for large-scale and open environments, such as the Internet.

Yu and Schmid [42] exploit roles assigned to agents to manage workflow processes. They traditionally model a role as a collection of rights (activities an agent is
permitted on a set of resources) and duties (activities an agent must perform). An interesting issue of this approach is that it aims to cover different phases of the application development, proposing a role-based analysis phase, an agent-oriented design phase, and an agent-oriented implementation phase. Anyway, Yu and Schmid do not take into particular consideration the implementation phase, suggesting the exploitation of existent agent platforms, which however do not implement role concepts. Finally, the fact that this approach focuses on the workflow management makes it quite close, but in our opinion there are some interesting cues (such as the exploitation of roles and the covering of different phases) that can be exploited in a wider range of application area.

RoleEP (Role based Evolutionary Programming) [38] aims at supporting the development of cooperative applications based on agents. In particular, it addresses applications where different agents collaborate to achieve a goal, defining an application as a set of collaborations among agents. RoleEP proposes four model constructs: environments, objects, agents and roles. A role is an entity that belongs to an environment and is composed of attributes, methods, and binding-interfaces. Role attributes and methods can be whatever needed to accomplish the related tasks. The binding-interfaces are exploited to dynamically associate role functions to objects; they can be thought as abstract methods, and are in charge of invoking the concrete methods of the objects they are bound. In this way, RoleEP enable agents to assume roles at runtime, granting high flexibility and enabling separation of concerns and role assumption at runtime. Nevertheless, RoleEP seems to focus on the implementation phase only, without providing support during the other development phases.

TRUCE (Tasks and Roles in a Unified Coordination Environment) is a script-based language framework for the coordination of agents [19]; the main requirements it aims to meet are adaptability, heterogeneity and concurrency. To these purposes, TRUCE allows the definition of coordination rules that are applied to roles that are assumed by agents, disregarding which actual agents will be implied in the coordination. In TRUCE, a role describes a view of an agent, hiding the other details, and letting the agent reveal only the properties concerned by the assumed role. The fact that TRUCE enables agents to adapt to various coordination protocols dynamically, makes it an open collaboration framework. As in the case of RoleEP, TRUCE addresses only the implementation level, by defining collaboration scripts that are interpreted by the agents.

The evaluation of the different approaches confirms that the exploitation of roles in agent-based applications provides different advantages. The main one is the separation of concern that they enable at different levels. In addition, they allow reuse of solutions, more agent-oriented views, and flexibility. However, none of the proposed approach exploits the concept of role to support the whole application development, leading from fragmented solutions. Starting from the above advantages and from the analysis of the limitations of the existing approaches, we propose the BRAIN framework detailed in the next section.

4 The BRAIN Framework

BRAIN (Behavioral Roles for Agent INteractions) is a role-based developing framework whose aim is to cover the agent-based application development process at different phases [10]. As shown in Fig. 1, the BRAIN framework provides three differ-
ent components, structured in as much layers: (i) a *model of interactions* that is based on roles, (ii) an XML-based *notation* to describe the roles, and (iii) *interaction infrastructures* supporting agents in the management of roles. Such infrastructures are based on the adopted model and rely on the defined XML notation. Our framework provides for more than one interaction infrastructure, since different environments can have different needs in terms of implementation, while it is important that all infrastructures relies on the same model and notation, so that high-level descriptions of solution can be easily shared and managed by developers.

![BRAIN framework](image)

**Fig. 1.** The BRAIN framework

### 4.1 BRAIN Model and Notation

In BRAIN, a role is defined as a *set of capabilities* and an *expected behavior*. The former is a set of *actions* that an agent playing such role can perform to achieve its task. The latter is a set of *events* that an agent is expected to manage in order to “behave” as requested by the role it plays. Interactions among agents are then represented by couples (*action*, *event*), which are dealt with by the underlying interaction system. Fig. 2 shows how an interaction between two agents occurs. When the Agent A wants to interact with the Agent B, it performs an action chosen among the capabilities provided by the role. This action is translated into an event by the interaction system, which is delivered to the Agent B, which is in charge of managing it in the appropriate way.

We chose this model of interactions because it is very simple and very general, and well suits the main features of the agents: the actions can be seen as the concrete representation of agent *proactiveness*, while the events reify the agent *reactivity*.

![Interaction system](image)

**Fig. 2.** The Interaction model in BRAIN
The notation proposed by BRAIN, called XRole [7], enables the definition of roles by means of XML documents. The use of XML grants high interoperability, since it leads to the capability of correctly parsing and understanding role definitions even by different software entities, possibly developed with different technologies. Furthermore, thanks to XML, developers can get different views on the same document, for example using a XSL style sheet. This leads to the capability of using only information really needed. An example of XRole document is reported in Subsection 5.1. It is worth noting that each different representation derives from the same information, so the different phases of the development of applications rely on the same information, granting continuity during the entire development. For instance, during the analysis phase, the analysts create XRole documents following the XML Schema proposed in BRAIN XML-based notation, which guides them in the definition of the role features. These XRole documents can be translated into HTML documents to provide high-level descriptions also for further uses. In the design phase, the same XRole documents can be translated into more detailed HTML documents to suggest functionalities of the involved entities. Finally, at the implementation phase, again the same XRole documents can be exploited to obtain Java classes that implement the role properties.

4.2 BRAIN Interaction Infrastructures

As shown in Fig. 1, BRAIN allows the implementation of different interaction infrastructures, which can be plugged-in into the agent system by the platform administrator. The interaction infrastructures enable the assumption of roles by the agents, possibly in different ways as shown in the following. Moreover, they manage the interactions between agents in the sense that they actually perform the translation of actions into events, and the delivery of the latter. The interaction infrastructures can control interactions and enforce local policies, such as allowing or denying interactions between agents playing given roles; this is done in addition to the security policies enforced by the adopted programming language, for instance the Java security manager.

This feature of BRAIN allows a great adaptability, since each platform can use the more efficient interaction infrastructure for its purposes. In fact, since we believe that there is not a single optimal implementation, the use of such architecture allows us to develop several implementations, testing them to find which are better and for which purposes.

It is important to note that the implementation infrastructure is the bottom layer of the BRAIN framework (see Fig. 1), and this means that the two top layers do not change depending on the interaction infrastructure. This leads to more reusable software, since all the analysis and the XRole documents can be applied to different interaction infrastructure implementation. In other words, the use of such architecture, allows the developers to concentrate on the analysis phase discarding each detail of the interaction infrastructure, because their results will be always portable on the BRAIN framework. A further consideration relates is the use of events in BRAIN, which enables different implementations to communicate using a common mechanism (i.e., events themselves). This is really important, since the two agents could be in platforms with different BRAIN implementation, and still able to communicate each other by exchanging events.
At the moment we provide two main implementations of the BRAIN interaction infrastructure, as shown in Fig. 1.

The first one, called RoleSystem [9], relies on abstract classes which represent roles. In this approach, when an agent wants to assume a role, it must register itself to the core of the system. When the RoleSystem core receives a registration request, it evaluates if the agent can assume the role and then, in case of success, returns to the agent a registration certificate. This certificate allows the agent to play the requested role, which means to execute role actions. In fact, the agent executes actions and waits for events through its registration certificate, which in other words is simply a connection between the agent and the role itself. When the agent decides to release a role it simply communicate to the RoleSystem core to dismiss the role. The core invalidates the registration certificate so that the agent can no longer use it, and this corresponds to the exit from the interaction context.

The main advantage of this approach is that it is quite simple, both to use and to implement. This leads to a quite light architecture which can be easily exploited from devices with limited resources (such as PDAs, mobile phones, etc.). Nevertheless this approach requires a central unit, the RoleSystem core, which overhangs all the role assumption/release process. This means that (i) there is a single point of failure in the whole system, and (ii) agents do not assume and release roles, but simply take connections to them.

Starting from the above consideration we developed a new and more innovative infrastructure, called RoleX (Role eXtension for agents) [5]. In RoleX a role is composed of a full-implemented class, whose members are added to the agent when it assumes the role, and of an interface, which agents can implement in order to appear as playing the assumed role (external visibility). RoleX relies on a special component, called RoleLoader, which performs bytecode manipulation fusing the agent and the request role in a single entity. After this, the agent can exploit role members, such as method or variables, in a direct way, since they are embedded in the agent itself. Furthermore, during the manipulation process, the RoleLoader forces the agent to implement (in Java sense) the interface of the role, so that the agent will appear, from an external point of view, as playing such role. In other words, no central services maintain the role played by the agent, because a simple instance will give the role itself. When the agent decides to release a role, the RoleLoader performs a new bytecode manipulation in order to discard all role members, so that the agent regains its original structure.

RoleX exploits better the XRole notation, using XML documents that describe each role in a finer grain than in RoleSystem (still keeping the two infrastructures compliant). In fact, in RoleX each role is described by nested sections of XML documents, called respectively RoleDescriptor, ActionDescription, and EventDescription. Each descriptor describes a particular behavior of the role, starting from its main aim (RoleDescriptor), each action the role provides (ActionDescription) and which events will be generated by such actions (EventDescription). Thanks to this structure an agent can better analyze available roles than in RoleSystem, choosing in a more dynamic way the one that better fits.

The main advantage of this approach is the high dynamism provided: the agent can choose more dynamically the role at run-time, thanks to the more modularized use of the XRole notation, and the assumption/release changes at run-time the agent behavior and appearance. Nevertheless, the use of the RoleLoader, which performs byte-
code manipulation entirely in memory, without source code alteration or recompilation, leads to the need of more resources than in \textit{RoleSystem}, and requires a more complex development.

Accordingly to the opening sentences of this section, we stress how is important to provide ad-hoc implementations dealing with the hardware-software context in which the BRAIN framework will be executed. Our implementations can cover both situations with limited resources (\textit{RoleSystem}) and where high dynamism is required (\textit{RoleX}). The two infrastructures proposed here are compliant with regard to the model and the notation, allowing also the communication between agents playing \textit{RoleSystem} roles and agents playing \textit{RoleX} ones.

5 An Application Example

This section provides an application example that exploits roles in order to build a more scalable and reusable large-scale software. The main aim of this section is to detail the steps involved in a role-prone development and to focus the advantages achieved by the use of roles.

The application chosen is a flying and hotel reservation system, developed of course as a MAS application in a large-scale environment. We suppose that reservations (for both flights and rooms) can be performed in appropriate Web sites that provide on-line services and can also host mobile agents. For instance, we can consider the hotel Web sites as the places where mobile agents can search for rooms and book them; similarly happens with the flight carrier Web sites for the flights. Our application exploits mobile agents capable of moving to different sites in order to search for rooms/flights and book them. In particular, we propose to split the application main task, which is the reservation of both the flight and the hotel for the user trip, into two smaller separated tasks: the reservation of the flight and the reservation of the hotel. Each task will be performed by a single mobile agent, which is in charge of traveling to find the best reservation for the hotel room or the flight\footnote{We do not specify what “best” means; it could be the lowest price, or the more luxurious class. For the purposes of this section it does not mind, and we assume that the agent knows how to identify the “best” solution, accordingly to the will of its user.}. The two agents will be called respectively \textit{hotelReservator} and \textit{flightReservator}. Note that in our example we will exploit only two agents for simplicity’s sake. In actual applications, a larger number of agents are involved: for instance, both the agents are likely to compete with other agents (belonging to other users) in order to obtain the reservation at a given price; another case is when the application is composed of different agents of the same kind, which perform a search in a parallel way. Nevertheless, the two agents of our example are representative of general interaction issues, and the adopted solution can be easily extended to more complex situations defining appropriate roles in a scalable way.

The application works as follows. As shown in Fig. 3, the two agents start from the user’s host, which can be a standard workstation, a laptop, a PDA or something else. After that the agents move around all available platforms searching for the reservation.

We have chosen this kind of application because it involves several requirements and coordination issues, which are typical of large-scale distributed environments. In particular it is important to note that:
Both the agents must interact with the external world, such as other agents or the reservation databases, in order to complete the reservation.

The two agents must interact each other, in order to reserve both the hotel and the flight in a coordinate way.

Please note that, since the two agents are autonomous, they could find the two reservations at different platforms. For example, with regard to Fig. 3, the hotelReservator agent can find its best reservation at the bottom right platform, whilst the flightReservator can find it at the bottom left one. In this case the synchronization between the two agents must be remote, which means it implies a remote interaction. Instead the interactions among the above agents and the external world (i.e., each platform) are always local. This implies that the two agents require a mechanism to find each other and to communicate, in order to synchronize the reservation. As detailed below, roles can be useful in this situation, leaving the agent developers free to reuse an already existent communication library.

Let us start our application analysis from the interaction among agents and the external world. Since both the agents must travel among different sites (platforms), and each site could have different ways to manage reservations, agents cannot embed in themselves the needed knowledge and behavior to face all possible situations. This suggests embedding this knowledge and behavior in a set of local roles, provided by each platform. Thanks to this, the agent developer is not in charge of knowing how the platform manages reservations, but it must simply develop the agents so that they will search (and play) an appropriate role. For example, as shown in Fig. 4, the hotelReservator agent can play a room_searcher role in order to search for and, in case, book the room. The hotelReservator agent is not in charge of knowing how the site manages reservations, since, as detailed above, the room_searcher role is in charge of knowing it. The agent simply assumes the role and then exploits a service via the role, which means the agent performs a role action. The role supports the agent in the execution of the service, translating the requested service into a site dependent action, and then returning results to the agent. For example, as shown in Fig. 4, the hotelReservator searches for all available rooms via a room_searcher role operation, and then it can evaluate the best (if there is one) for the user. Fig. 4 a) shows how the room_searcher role performs a direct query to a local database, while the figure b) shows a more complex situation, where there is an administrator agent which manages and authorizes all accesses to the database. The difference in the above cases is
that in the former the access to the booking database is direct, while in the latter it is undertaken to the will of the administrator agent. Nevertheless, as written more times, the hotelReservator agent does not worry about the access way to the database, since it is the role that translates the requested service to the site dependant action(s).

![Fig. 4. Exploiting a role to perform different platform specific actions](image)

Similar consideration can be done for the flightReservator agent, which can exploit a flight_searcher role to query the flight-booking database. The use of the above roles allows the agents to better interface to a lot of different scenarios, but not only: since these roles are provided by each platform, the agents are smaller, in terms of code size, making faster the mobility (i.e., agent transmission through the network).

The above roles can be played by the agents to book effectively a flight or a room, when they judge that the reservation is “good”. But as written above, one requirement of this application is to reserve a room in a coordinate way with the flight reservation. To achieve this goal the agents must interact each other. For example, when the hotelReservator finds the best available room, it must communicate to the other agent, the flightReservator, that the room has been found, so that starting from this moment the other agent can book the flight. Furthermore, when the flightReservator find the best flight, must contact again the hotelReservator in order to check if the room is still available (since the hotelReservator is waiting for a flight to be found), and if both the agents can successfully book the service, they synchronize themselves and complete the booking. Of course similar consideration can be applied in the case the flightReservator finds the flight before the hotelReservator finds a room. Both the agents can book the room or the flight using a time-out. They have to confirm the reservation before the time-out expires, else the room (or the flight) will be considered available again. The time-out is required to avoid deadlock problems and to grant to an agent the required time to contact and deal with its partner (i.e., the other agent). For example, supposing that the hotelReservator finds the best room, it can temporary lock it, and then it can contact the flightReservator in order to establish if the room can be definitively booked. If the flightReservator has found the best flight, both the agent can book; if the time-out expires, the application continues. The use of time-out partially solves the communication between the two agents, which is still a distributed-commit problem.

The flow chart in Fig. 5 shows the application logic needed to achieve this synchronization. From that state chart it is possible to note how, when an agent finds the best reservation, it asks the other agent if it has found the best reservation too (and if
it is still available). If this is the case, the booking can be committed for both the agents. In the other case, the agent must wait until the other agent finds the best reservation (i.e., it waits for a message). When this happens the booking can be committed. Please note that, since the first agent that finds the best reservation must wait until the other finds it too, the reservation could be no more available (maybe booked by an agent of another user). It is for this reason that the agents must synchronize themselves also on the availability of the reservation.

The above considerations suggest the use of a quite complex mechanism to synchronize the two involved agents. Instead of developing this communication mechanism, which implies also the use of a way to track the agent position (i.e., on which host the other agent is), developers can still use ad-hoc roles. The problem of finding another agent running in another platform/site is a quite common problem in all mobile agent applications, and different communication protocols have been built to allow it. Instead of embedding each possible discovering/communication protocol, such as MASIF [33], RMI [35], Jini [23], CORBA [12], JXTA [24], etc., in the agent, this can exploit a remote_communicator role to discover and communicate (i.e., exchange messages) with it. This role hides all network specific details, even about proxies and protocol tunneling. Fig. 6 briefly shows how the agents can communicate exploiting the above role. Note that the agents can assume more roles at the same time, such as the one required for the remote communication and the one required for completing the booking. Similarly an agent has to assume the remote_communicator role only when this is strictly needed, that means when a (possible) reservation has been found. This emphasizes how roles provide dynamic services, which can be used at run-time on demand.

When both the agents have found their reservations and have rightly booked the room and the flight, they return the results to the user and the application ends.
The above application has shown how useful can be roles in the development of large-scale mobile-agent applications. In particular, roles allow developers to not worry about specific platform-dependant details, providing also code reusability, since the same role can be exploited from different agents in different applications. Furthermore, since roles act as a run-time addition, they can be developed in a separate way from agents, enabling separation of concerns that simplifies application development. This implies also that roles can evolve separately from agents, easing maintenance and modification of the software; agents can continue to play roles without worrying about these changes.

5.1 Application Code

This section shows some pieces of code related to the above detailed application; in particular we report the definition of roles in XRole (independent of implementation), and the implementation of roles in the RoleX infrastructure.

Fig. 7 shows the main part of the XML notation related to the hotelReservator role. The document is compliant with the http://polaris.ing.unimo.it/schema/RoleDescriptionSchema schema, which is not detailed here due to space limitation. The document can be read by humans, and it defines a role through a role descriptor (tag <GenericRoleDescription>), which defines some semantic data related to the role itself. This data can be, for example, a list of keywords, an aim, the version, etc. Nevertheless, even if this semantic data is really useful to search for a particular role and analyze it, it is not enough. In fact, since an agent will use some role services (i.e., operations) to perform its task(s), the description of these services must be included in the XRole document. To achieve this, the role descriptor can contain one or more operation descriptors. An operator descriptor describes a single role operation that the agent can use to perform a specific task; this information is embedded in the tag <OperationDescriptor>. As for the role, an operator descriptor collects data such as keywords, aim, name, version, etc. and a piece of information that enables the agent to actually request the service to the implementation infrastructure. In particular this binding is achieved through the method name and the events related to the method (respectively with the tags <methodName> and <receivingEvent>). The former information details which method will be invoked to perform the specific operation, while the latter details which event(s) will be sent from the above method.
Role-Based Approaches for Engineering Interactions  

With regard to the **hotelReservator** role, Fig. 7 shows the two main operations of the role itself. The first, realized through the method **ListRoom**, simply returns a list of available rooms at a particular host/platform. The second, based on the method **ReserveRoom**, can be used to reserve a specific room. This method accepts also two parameters, both of type integer, that are the credit card number used to confirm the reservation and the room number. The return value, that is an integer too, is the reservation number id.
Fig. 8. The Java code for the hotelReservator role

There is an important difference between the above two operations: the latter, which corresponds to the ReserveRoom method, declares also an event, called ReservationConfirmEvent, while the other operation does not. The above event is sent back from the system to the agent to inform that its reservation has been accepted. The
agent must manage this event in order to know its reservation status\(^2\). Since there is no need for the ListRooms method to get back an event, because it is a simple query to the system, it makes sense for the ReserveRoom since the reservation could require some time to be complete, and during this time the agent should be free to do something else (e.g., contacting the other agent).

After the XRole document, let us consider how to implement a role compliant to it. Fig. 8 shows an example of a Java implementation for the RoleX infrastructure.

The two methods ResultSetToInteger and ResultSetToList are not explained here because their implementation is not important for the purposes of this paper.

The role shown in Fig. 8 is thought for a platform where rooms are managed through a relational database. Nevertheless, as already written in the previous sections, roles allow agents to not worry about such details, so that an agent has simply

---

\(^2\) We simply suppose that the ReservationConfirmEvent can carry either a confirmation or a reject, so that a single event is needed to inform the agent about its reservation status.
to use the \textit{hotelReservator}, independently of its implementation. For instance, if the reservations are stored in a file, the code of the role must be adapted, but the interface remains the same and the agent must not be modified.

The last piece of code shown in this section is related to the agent that is going to play the above role. Fig. 9 shows an example of Aglet agent, that is an agent for the IBM Aglets platform [29], which exploits the above \textit{hotelReservator} role. In the reported code, the agent knows the name of the role, but actually the RoleX infrastructure exploits role descriptors to uncouple the needed role from its implementation. In this way, agents can provide keywords to search for roles [5]. The role descriptors are not detailed here due to space limitation.

After the agent has asked the RoleX bytecode-manipulator, called \textit{RoleLoader}, to load the specified role (i.e., to add the role members to the agent itself; for further details see [5]), it starts using the role through the special method \textit{act(..)}. The latter is an added method that the agent can use after a role assumption (i.e., this method is added by the \textit{RoleLoader}). This method performs a kind of reflection in order to find the role operation, and then invokes it and returns back the result.

Please note that the agent, after the role assumption, register itself as an \textit{event listener}. This means that the RoleX implementation scatters generated events to that listener, which is in charge of managing them. In connection to the application example, when the booking system wants to confirm (or reject) the reservation, generates an event (i.e., a Java objects that contains a specific payload), and the agent, since it has registered itself, receives the event and manages it.

\section{Conclusions}

This chapter has presented how the concept of role can be fruitful exploited in engineering agent interactions in large-scale applications. Roles can carry different advantages, in terms of separation of concerns between the algorithmic issues and the interaction issues, generality of approaches, locality, and reuse of solutions and experiences. In dealing with agent interactions, roles can simplify the development because can embody the interaction issues, which are managed separately from the algorithmic issues that are embodied into agents. Roles can be developed separately from agents, implying separation of issues, solution reuse, more scalability and easier maintenance.

The survey of different existing approaches has confirmed such advantages in the development of agent-based distributed applications, but has also outlined a high degree of fragmentation in the evaluated approaches. In fact, the concept of role is usually exploited in only one phase of the application development, while only few approaches take roles into consideration in more than one phase.

Starting from the above consideration, we have proposed BRAIN, a framework to support agent developers during their work. BRAIN exploits the concept of role in different phases of the development, since it is based on a simple yet general role-based model for interactions. The presence of an XML-based notation enables interoperable and flexible definitions of roles, which can be tailored to different needs and managed by humans, agents and automatic tools. Finally, interaction infrastructures based on the BRAIN model and adopting the BRAIN notation provide for a suitable support at runtime. An application example has shown how BRAIN can be exploited in a large-scale agent-based application.
With regard to future work, we are exploring some directions. First, we want to evaluate our approach in different application areas; we are currently developing a role-based application for e-democracy, to support the participation of citizen in public life. Second, we are going to consider trust and security issues in the use of roles: this will make our approach more feasible for real applications. Finally, we are exploring the development of high-level support for specific phases of the application development, built on top of the XRole notation.

Acknowledgments

Work supported by the Italian MIUR and CNR within the project “IS-MANET, Infrastructures for Mobile ad-hoc Networks”, and by the MIUR within the project “Trust and law in the Information Society. Fostering and protecting trust in the market, in the institutions and in the technological infrastructure”.

References


Evaluating Agent Architectures:  
Cougaar, Aglets and AAA

Ian Gorton, Jereme Haack, David McGee,  
Andrew Cowell, Olga Kuchar, and Judi Thomson

Information Sciences and Engineering  
Pacific Northwest National Laboratory  
Richland, WA 99352, USA

Abstract. Research and development organizations are constantly evaluating new technologies in order to implement the next generation of advanced applications. At Pacific Northwest National Laboratory, agent technologies are perceived as an approach that can provide a competitive advantage in the construction of highly sophisticated software systems in a range of application areas. To determine the sophistication, utility, performance, and other critical aspects of such systems, a project was instigated to evaluate three candidate agent toolkits. This paper reports on the outcomes of this evaluation, the knowledge accumulated from carrying out this project, and provides insights into the capabilities of the agent technologies evaluated.

1 Introduction

Agent technologies are a potentially promising approach for building systems that require intelligent behavior from a collection of collaborating, autonomous software components. With their roots in the field of distributed artificial intelligence [1], agent technologies promote a variety of abstractions, protocols and mechanisms that aim to make it easier to construct distributed, collaborating systems [2]. While some of these technologies have been productized, many remain the outputs from various leading research laboratories and universities. This creates a complex landscape for organizations wishing to exploit agents in their applications.

Researchers in the Information Sciences and Engineering division and elsewhere at the Pacific Northwest National Laboratory (PNNL), have considerable interest in using agent technologies across a number of application areas. These application areas include next generation energy management systems, mining information from massive data sets, intelligent multimodal interactive systems, and cyber security. While each application area has a different set of requirements for the agent technologies, the common thread is the need for distributed, collaborating software components that collectively behave in a goal-driven manner.

Consequently, we evaluated a number of agent architectures that were likely to be of relevance to us. The three architectures were chosen based on the requirements for
agent technologies of the PNNL research groups, based solely on their business needs and existing research collaborations. While it would have been desirable to evaluate other technologies such as FIPA-OS, ZEUS and Jack, we could not provide a business justification at the time the project commenced.

Selecting appropriate technologies and architectures for a given application domain is a key activity in the software engineering process [3, 4]. In practice, many projects fail or greatly exceed budget because they base their solutions on technologies that do not perform as anticipated.

The aims of this evaluation project were as follows:
Qualitatively investigate various features of each technology to assess capabilities against the requirements of the various application areas.
Quantitatively assess each technology in terms of its performance and scalability.
Assess the potential for code reuse and interoperability between applications written using different agent infrastructures.

The three technologies chosen for evaluation were AAA [5], Aglets [6] and Cougaar [7]. This represents a mix of a production-ready technology (Cougaar), an established agent technology research vehicle (Aglets) and a relatively new technology from a leading research university (AAA). This provides a further dimension of interest in the comparison. The evaluation consisted of constructing an identical example application with each technology, and then comparing those applications both qualitatively and quantitatively. The results of the evaluation reveal several important differences in the capabilities of the three agent systems. The results also highlight some of the inherent complexity of the technologies, the variable levels of implementation maturity, and the different performance characteristics.

The following sections briefly describe each of the technologies evaluated, the test application architecture, and the architecture and design used to create the test application with each technology. The comparative results are then presented, including the performance achieved on a consistent test-bed, and an assessment of some of the important features of each agent platform. The paper concludes with a discussion of related work and an evaluation of the effectiveness of our approach.

2 Technology Overviews

2.1 Aglets

The Aglets software development kit and framework were originally devised at IBM’s Tokyo Research Laboratory. The source code was subsequently released as open source available under the IBM Public License. The fundamental differentiating technology in the Aglet architecture is agent mobility.

The Aglet object model (AOM) is designed to exploit the strengths of Java, including platform independence, secure execution, dynamic class loading, multithreaded programming and object serialization. At the same time, Aglets provide additional
capabilities for features such as resource control, protection and object ownership of references, and support for preservation and resumption of execution state.

The basic elements of the AOM comprise three key abstractions.

**Aglet:** a mobile java object that can move around aglet-enabled hosts within an environment.

**Proxy:** a representative of an aglet that serves as a shield, protecting direct access to the public methods and providing location transparency for the aglet.

**Context:** the aglets workplace, a stationary object that provides a uniform execution environment. Many aglets can exist in a single context, and a single computer may run multiple contexts.

The aglet package consists of a set of Java API’s and a run time environment. Each aglet is a separate class - a master class is written and loaded into the Aglet Tahiti server where the aglet is created and begins execution. For mobility, each machine must be running a Tahiti server.

An aglet has a number of fundamental operations. There are only two ways in which an aglet can be created – it may be instantiated (created) or it may be copied from another aglet (cloned). Aglets may also be removed when they are of no further use (disposal). Aglets can also be temporarily halted (deactivation) and restored (activation) if required. Finally, an agent can be pushed (dispatched) to a new destination context (local or remote) and retrieved from that context (retraction).

A messaging framework supports inter-aglet communication. The principle method for aglet communication is through message passing. A simple event scheme means that it is only necessary to implement message handlers in an aglet for those kinds of messages that are expected. A message is a type of object, characterized by its ‘kind’, denoted by a string. In addition, atomic arguments may be attached or multiple arguments wrapped as an object. The object-based messaging framework provided is location-independent, so that two aglets need not exist within the same context or in the same location to communicate. Both synchronous and asynchronous messaging is provided; blocking operators may halt execution of an aglet until its companion replies to the message, or execution can continue through the use of a dummy object. Aglets also support multicasting in a publish-subscribe manner. Aglets can subscribe to one or more multicast messages and listeners for those messages are then implemented to handle occurrences.

### 2.2 Cougaar

Cougaar is a Java based agent architecture developed for DARPA under the Advanced Logistics Program. It is designed to solve logistics planning problems using a cognitive model based on human reasoning. Cougaar is an open source technology implemented as a set of Java APIs.

A Cougaar agent consists of a blackboard that facilitates communications (known as the *plan*), and operations modules called *plugins* that communicate with one another through the blackboard and contain the logic for the agent’s operations.
Cougaar agents may be created statically using configuration files, or dynamically using an Agent Server. The server publishes an interface that allows other agents and components to create, destroy, or reconfigure agents within its boundaries. Regardless of the creation method, all agents have a set of plugins and a defined place in the agent community that declares the agent’s name, roles, and relationship to other agents.

In addition to a blackboard that is internal to each Cougaar agent, the architecture supports one-way, asynchronous, and one-to-one communications through a MessageTransportServer. This server, along with the NameServer, allows a message to be sent to an arbitrary agent based on its identifier. Finally, although agents may not directly address the blackboard of another agent, two agents that have a defined relationship may share information directly by acting as Organizational Assets for one another.

Cougaar supports inter-agent communications through a mechanism known as allocation. An agent will create a task and then allocate this task to an asset. If the asset is another agent, then the task will appear on its blackboard where a plugin subscribing to the task will be notified. The receiving agent then performs the behavior required and sends the results back by changing the status of the task allocation’s results, which causes a notification to be sent to the publisher.

The plugin execution model was designed so plugins can run in parallel. In practice however they run using a shared thread. A PluginScheduler decides when an individual plugin should run. Plugins must explicitly spawn separate threads for functionality that may occupy the shared thread for extended periods of time.

2.3 Adaptive Agent Architecture (AAA)

The Adaptive Agent Architecture (AAA) is a multi-broker, multi-agent system architecture. AAA was developed in the Center for Human Computer Communication at the OGI School of Science and Engineering of the Oregon Health & Science University to promote active research in multi-agent systems semantics and communication. It extends and improves upon version 1.0 of the Open Agent architecture developed at SRI.

Major goals for the AAA are:

- Support for a large variety of hardware platforms, operating systems, and programming languages via a lightweight API.
- The use of Speech Acts semantics driven communication primitives that are passed as messages in the AAA Agent Communication Language (ACL), which is backwards compatible with OAA version 1.0 and provides a more thorough semantics for middle-agent and group communication [8].
- An easy-to-implement Java Interface that automates the process of converting Java data types and collections into an ACL-supported content language such as Prolog Horn clauses or XML.
• Fault-tolerance achieved through the execution of the joint commitments established via the Speech-Acts based communication, resulting in the ability of Facilitators (or other agents) to re-establish connectivity with agents that were registered with another Facilitator, when communication issues arise [8]

• A blackboard and accompanying algorithms for unification of terms on them, which can be used to create agents with a persistent data store. The Facilitator uses a blackboard internally in order to implement its message passing architecture, and agents can use their own blackboards for any purpose.

On initialization, AAA agents register their capabilities with the Facilitator. Messages are brokered by the Facilitator based on the registered capabilities. After an agent has connected to the Facilitator and registered its capabilities, it can then communicate with other agents in any of four ways, as follows:

• **Requests:** These can be either synchronous (the agent thread blocks until the reply returns) or asynchronous (a callback is registered to listen for the reply and is called when the reply arrives). Requests are usually brokered by the Facilitator and routed to capable agents. The Facilitator can accumulate replies, manage distribution of complex action requests to capable agents (agent A do X or agent B do Y), and insure deadlines are met.

• **Informs:** Declarative statements can be made by any agent. If these are sent to a Facilitator then the middle agent will transmit the *informs* to any agent that has registered similar interests.

• **Direct:** Agents may request a list of capable agents from the Facilitator, who replies with a list of agent addresses. The agent can then communicate directly with any agent in the list without Facilitator intervention. Such communication, once brokered, can remain in AAA ACL (i.e., Requests and Informs) or the agents can negotiate a higher performance protocol (e.g., sending Java Objects or other binary directly).

### 3 Test Scenario and Process

In order to compare and contrast these three technologies, a test scenario was devised, which could be implemented with each. This scenario would provide a basis for direct comparison. The main objectives of the test scenario were to facilitate the:

1. Comparison of the architectures that agent technologies support/promote
2. Comparison of the performance of the agent technologies under test

The scenario is based upon a synthetic stock purchasing application, in which a multi-agent implementation intelligently decides which stock items to purchase from a set of proposed purchases and a budget. An overview is given in Figure 1.

The input is a series of discrete stock purchase requests, where each request comprises a monetary amount, the budget, and a collection of up to 10 individual stock items that the client wishes to purchase. Stock items have a key, ranging from 1 to...
1000, and a current price. A pool of 20 expert agents is dedicated to maintain expert knowledge about a portion of the range stocks. When asked by another agent, expert agents will reply with either a ‘buy’ or a ‘don’t buy’ message.

Each expert agent is configured to provide advice on 1/10th of the total population of stock, and hence the test is designed with some overlap between the areas of expertise of the experts. For example, one expert agent can intelligently answer questions about items 0-99, another about items 50-149, another about 100-199, and so on.

A purchase agent inputs a new purchase request, looks at the budget and calculates the total cost of the purchasing all the stock requested. It then optimizes the purchase in terms of the absolute number of requested stock that can be bought for the budget. Once it has decided which stock it will attempt to purchase, it sends a request to an expert agent that has knowledge about the quality of that stock (i.e. whether it’s a good buy at this particular moment). The specific expert agent is selected from the ones available using some form of directory service, in which agents advertise their abilities.

Fig. 1. Test Scenario

The expert agent receives the request, and returns a message that either confirms a buy action, or indicates that this item should not be purchased. Internally, the expert agent randomly computes 80% ‘buy’ responses, and 20% ‘don’t buy’ responses.

Depending on the responses of the expert agents, the purchase agent again attempts to optimize the purchase request, eliminating the latest ‘don’t-buy’ opinions. This may require further communications with other expert agents to confirm that another stock item purchase is sensible. At some stage, potentially after several iterations, the purchase agent will have a recommended set of stock to purchase. It then passes this result set to the buying agent. In this test scenario, the buying agent maintains statistics on the results for benchmarking purposes.
In order to provide a solid basis for performance comparison, a collection of 10,000 randomly created purchase requests were created. A solution to the test scenario was built with each agent architecture and installed on the same high-performance Windows 2000 machine. The multi-agent system solution for each architecture ran on that system in one Java VM.

The same Java class was used in each solution to read the 10,000 purchase requests from a file and make these available to the purchase agent. The same code was also used to implement the expert agent behavior (80/20 random Boolean response) across solutions.

To measure performance, the purchase agent records the number of iterations needed to finalize a purchase. The buying agent also measures and records the time taken to produce each purchase. These measures are stored in memory and output to a log file at the end of the test, along with the total elapsed time to process the 10,000 inputs.

4 Solution Implementations

The following describes the three implementations. Each implementation was written by the same software engineer, and much code was reused across implementations to ensure as much consistency as possible.

4.1 Cougaar Architecture

The Cougaar solution comprises 2 agents, as depicted in Figure 2. The Purchase agent exploits Cougaar’s blackboard mechanism to load all the purchase orders (POs) on to the blackboard, which is shared by 4 agent plug-ins. When a new PO appears on the blackboard, the Plan plug-in is notified. It selects which stock to attempt to buy given the budget, and places the result on the blackboard. This causes a notification to be sent to the Ask plug-in. It selects an appropriate Expert agent for each stock in the requested list, and allocates a task to the Expert, which causes it to appear on that agent’s blackboard.

When the Expert agent sees the task, it responds by updating the allocation results on the Purchase agent’s blackboard. This causes a notification to be sent to the Ask plug-in. Once all the Experts respond, the Ask plug-in notifies Plan, and it decides whether the purchase is complete, or needs to be altered due to ‘don’t buy’ responses from Experts. In the former case, Plan sends the completed order to the Buy plug-in. In the latter, Plan alters the selected stock list, and the whole process of asking Expert agents occurs again.

The evaluation of all the POs occurs concurrently. Each agent has its own thread, and all the agents run in the same JVM. An agent’s Plug-in Scheduler coordinates the execution order of plug-ins in an agent. This order is non-deterministic, depending on the order of messages and notifications that are passed between the Purchase and Expert agents, and the plug-ins.
4.2 AAA Architecture

In the AAA solution, the agents reside in the same JVM and after the Facilitator brokers connections between the Plan and Expert agents, these agents are configured to communicate directly, which is potentially more efficient than using the AAA Facilitator to broker all communications.

The Load agent sends a direct inform to the Plan agent for each PO. Plan selects stock items to buy, and send an inform message to the Ask agent. On start-up, Ask sends a series of request messages to the AAA Facilitator to get references to each of the Expert agents. It then stores these locally in a data structure, and uses this to select an expert to evaluate each stock item. This solution is depicted in Figure 3.

Ask creates a thread for each stock item in the request, and these threads send request messages to their allocated Expert agent. Each Expert’s reply is handled by a
AAA callback (*NotificationListener*), and when all replies are received, *Ask* sends an inform message to *Plan* that contains the results. If the order is complete, *Plan* sends an inform message to *Buy*. If the order needs work, *Plan* selects different stock items and cycles again through the stock evaluation process.

Since it is not possible to publish Java Objects on the AAA’s shared blackboard, Java objects, such as purchase requests, must be converted into a supported content-language format, such as XML. Although a framework and helper classes exist to facilitate this conversion, due to time constraints we decided to implement the AAA solution as described above.

### 4.3 Aglets Architecture

The architecture for the Aglet’s solution is depicted in Figure 4. As Aglet’s does not support a shared blackboard, the application is structured as a set of communicating aglets, each of which has its own thread of execution. The *Load* aglet sends the input stream of POs to the *Plan* aglet. This is an asynchronous one-way communication. *Plan* inputs each PO and creates a list of selected stock items, which it asynchronously sends to *Ask*. *Ask* sends an asynchronous request to the *Expert Coordinator*, which replies with references to the appropriate *Expert* aglets. *Ask* then spawns a set of threads, one for each stock item, and these threads each communicate asynchronously with an *Expert*. When all the threads receive a response, the *Ask* aglet asynchronously notifies *Plan* about the outcomes. *Plan* responds accordingly by either asynchronously sending the completed PO to *Buy*, or selecting new stock and repeating the *Expert* evaluation process.

![Fig. 4. Aglets Solution](image)

The solution co-locates all these aglets in the same JVM. In order to support asynchronous communications, aglets have request queues in which pending requests are placed. Hence the *Load* aglet effectively places all the POs in the request queue for *Plan*, which processes them serially and places the results in the *Ask* aglet’s request queue. From there, the processing hence proceeds non-deterministically due to the threading internally in *Ask*, contention for *Experts*, and the asynchronous notification mechanism used between *Ask* and *Plan*. 
5 Evaluation

5.1 Technology Evaluation

As expected, each of the three technologies proved to be rather different in nature.

Table 1 briefly evaluates each against a core set of review criteria that we devised. These criteria were based on extensive technology and architecture evaluation experience, as reported in [3, 4], and have considerable overlap with the review criteria in [14]. We focused on a set that covers key architectural features and flexibility, as well as robustness and quality of supporting documentation. These were deemed the most important criteria for the research groups that were examining agent-based approaches.

- **Documentation**: How easy was it to learn the technology using the documentation? Was the documentation comprehensive, current and easy to navigate?
- **Scalability**: Can the technology exploit multiple processors on the same machine (scale-up), and a fully distributed deployment over multiple machines (scale-out)?
- **Ease of modification/distribution**: How easy is it to modify the solution/code to re-factor the functionality across different agent collections, or to physically locate agents in different locations? Does the technology support location transparency?
- **Architecture flexibility**: Does the technology promote a particular architectural style, such as shared blackboard, distributed memory, or publish-subscribe? Or does the technology provide flexibility in terms of matching the architecture to the problem domain?
- **Mobility**: Does the technology support mobile agents?
- **Inter-agent communication mechanisms**: Does the technology provide a flexible and comprehensive set of inter-agent communication mechanisms, including synchronous, asynchronous and multicast?
- **Robustness of implementation**: Did the technology fail during development or testing? Were certain mechanisms prone to failure?

In terms of raw performance, formal performance testing was carried out with each solution. The final tests executed the solutions 5 times on the same hardware platform. Results were averaged and analyzed to ensure that each solution experienced similar amounts of iterations in optimizing a purchase request. Table 2 presents the average performance measured in seconds for each solution to process 10,000 purchase orders, and the throughput obtained in messages per second.

Cougaar’s performance clearly benefits by being able to provide a solution to this problem using a blackboard-based architecture. Aglets provides just under 50% of Cougaar’s performance. This is still a relatively impressive result given the non-shared memory solution. AAA suffers from the fact that its inter-agent communications mechanisms are not optimized for the case when agents are co-located in the same JVM. Passing and decoding messages as Prolog Horn clauses also incurs additional performance penalties.
### Table 1. Qualitative Technology Comparison

<table>
<thead>
<tr>
<th>Criteria</th>
<th>Cougaar</th>
<th>Aglets</th>
<th>AAA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Documentation</td>
<td>Very extensive documentation that is current and well maintained</td>
<td>Extensive documentation, adequate for most purposes.</td>
<td>Documentation is incomplete and out-of-date</td>
</tr>
<tr>
<td>Scalability</td>
<td>Solutions should scale across multiple machines. 1 thread per agent restriction inhibits internal agent concurrency.</td>
<td>Designed to scale seamlessly across multiple nodes that support Aglet container.</td>
<td>Solutions inherently support distribution. Replicated facilitator mechanism should make a large-scale deployment possible</td>
</tr>
<tr>
<td>Ease of modification and distribution of solutions</td>
<td>Not simple to refactor solutions by modifying structure of plug-ins and agents. Code changes required.</td>
<td>Not simple to modify aglet structure, as sender must know explicit network location of destination aglets as there is no name service.</td>
<td>There should be no code changes necessary as all inter-agent communications must exploit distributed communications mechanisms.</td>
</tr>
<tr>
<td>Architecture flexibility</td>
<td>Plug-in architecture suggests an architecture style based on extensive use of shared blackboards</td>
<td>Aglet architecture forces fully distributed architectural style.</td>
<td>Flexible in terms of ability to easily mix shared blackboard and distributed inter-agent communications architectures. Moderately supported. ACL messages carry code and data (i.e. state).</td>
</tr>
<tr>
<td>Mobility</td>
<td>Supports agent mobility. A Plugin within an agent must decide when to issue the move, which agent should be moved (potentially itself) and which node it should move to.</td>
<td>Inherently supported</td>
<td>Supports synchronous and asynchronous and multicast inter-agents communications.</td>
</tr>
<tr>
<td>Inter-agent communication mechanisms</td>
<td>Excellent support for flexible publish-subscribe based messaging based on changes on blackboard state.</td>
<td>Excellent support for synchronous, asynchronous and multicast inter-agents communications.</td>
<td>Supports synchronous and asynchronous inform and request speech acts, which can be direct or indirectly managed through the facilitator.</td>
</tr>
<tr>
<td>Robustness of technology implementation</td>
<td>Robust and reliable</td>
<td>Robust and reliable</td>
<td>A few advanced mechanisms are ‘buggy’</td>
</tr>
<tr>
<td>Agent Technology</td>
<td>Overall test time (secs)</td>
<td>Throughput (POs/sec)</td>
<td></td>
</tr>
<tr>
<td>-------------------</td>
<td>--------------------------</td>
<td>----------------------</td>
<td></td>
</tr>
<tr>
<td>Cougaar</td>
<td>35</td>
<td>286</td>
<td></td>
</tr>
<tr>
<td>Aglets</td>
<td>76</td>
<td>132</td>
<td></td>
</tr>
<tr>
<td>AAA</td>
<td>500</td>
<td>20</td>
<td></td>
</tr>
</tbody>
</table>

It would be interesting to experiment and observe how these architectures could scale out to execute on multiple machines. Although we have not had time to fully test a distributed solution, preliminary tests indicate that the results would be somewhat closer in terms of throughput. Also, as indicated in Table 1, the Cougaar solution would require considerable code changes to spread the processing of the *Purchase* agent across multiple nodes. In contrast, both the Aglet and AAA solutions can be distributed with relative ease.

**Diversity:** The lack of standard approaches in the agent arena is starkly exposed in these three toolkits. Cougaar has a somewhat eclectic architecture, in which generic agent classes are endowed with specific behaviors through *plugins*. This architecture has been realized due to the evolution of Cougaar over a number of years. The AAA and Aglets take a more traditional approach to extending Java to incorporate agent features. However the precise semantics and mechanisms used for agents and communications are very different. This makes it a non-trivial task for a developer to understand and move between different technologies.

**Maturity:** Cougaar has a considerable infrastructure that must be understood before applications can be successfully built. However the documentation is excellent, and the technology seems to be well implemented. AAA suffers from inadequate documentation, which makes it difficult to build applications whose behavior deviates too far from the examples provided. The solution to this problem was to use Internet mailing lists and to communicate with the developers of the toolkits. Aglets was especially difficult to configure and get running, but it proved reliable once ‘tricks’ had been learned. However part of the reason for this was its ability to support mobile agents, a feature that was not exploited in this project.

Cougaar’s extensive infrastructure also provides more features that are useful in application deployments. Its ability to recover from failures at the local agent level is particularly useful. AAA has support for replicated *Facilitators*, which eradicates a single point of failure in an application. This however comes at the cost of state replication across *Facilitators*, which may not scale to large applications. At the cost of increased communications overhead, AAA also provides a rich ACL-based communications mechanism, which aims to simplify the implementation of complex communication schemes.

Aglets provide a lightweight support infrastructure, but add the ability for agents to move from node to node in an Aglets application. This is a potentially useful feature in a range of applications, and one that is not as well supported by Cougaar and the AAA.
5.2 The Evaluation Process

The following summarizes the strengths and weaknesses of the evaluation process that was used:

**Cost:** The project consumed 700 hours of effort in total. 130 hours were spent on project design, management and writing up the results. The remaining 570 hours were divided almost evenly between the engineers working on each technology to design, build and test the solution. Hence it took approximately 190 hours of effort to perform each investigation.

**Effectiveness:** The cost of learning the different technologies inevitably consumes a significant part of the budgets of evaluation projects such as this. With limited budgets, this makes it extremely difficult to carry out as thorough an investigation as pure research curiosity would dictate. However, taking a practical approach and gaining development experience across all three technologies creates a basis for rational comparison of the architectures and features. The development effort forces engineers to obtain a deep understanding of the features, and hence a higher quality evaluation is possible.

Measuring performance also provides insights into the quality of a technology. Performance is a key requirement in many modern distributed agent applications, and the test results give some concrete indications on the relative performance that can be obtained with each technology. Performance measures can also focus analysis on how a technology is likely to scale. Potential bottlenecks, poorly designed or implemented infrastructure components and non-scalable architectures quickly become apparent when a test application is available for performance tuning.

In summary, evaluation projects such as this one are typically a balancing act between what is desirable versus what is achievable with the budget available. In this project, the difficulty of learning some of the agent toolkits meant we slightly exceeded budget, mainly because of exploring implementation alternatives.

6 Related Work

As agent technology matures, it is being applied to produce solutions to real problems in many application domains. However, agent based systems have been typically implemented with ad-hoc solutions (communication languages, protocols, etc.) to meet the problem domain requirements. In response to this situation, standards are being created and implemented, such as Foundation of Intelligent Physical Agents (FIPA) [9]. As noted in [10], only recently have the FIPA standards been applied to practical applications. This cited work presents an analysis and evaluation of certain aspects of the current FIPA standards on the basis of the experience gained in developing a video entertainment broadcasting system. The authors analyzed the pros and cons of using FIPA standards for building MAS for such systems.
Performance and evaluation papers have recently started to appear in the literature. Many papers have been written on evaluating certain techniques, learning paradigms, scheduling mechanisms, and other software components that are important to the development of a MAS. NASA Goddard Space Flight Center has started to develop an Agent Concepts Testbed (ACT). ACT [11] is an environment in which rich agent and agent-community concepts can be evaluated through detailed prototypes and scenarios.

In several papers, authors have started to evaluate overall agent architectures, not just sub-components. Yamamoto et al. [12] report on a performance evaluation of a MAS based on large numbers of agents that process jobs by interacting with each other in an electronic commerce environment. The authors focus on evaluation of controlling the memory and CPU usage in a MAS containing thousands of agents. Performance was evaluated based on measuring throughput for two tests (single-server system and two-server system). They noted that by scheduling the activities of agents in an appropriate way, the throughput of agent interactions can be kept to a constant value in response to an increase in the number of consumer agents.

A recent study [13] reports on a direct comparison between the performances of platforms in mobile agent systems. The authors present results of a benchmarking study comparing eight Java-based mobile agent systems (Aglets, Concordia, Voyager, Odyssey, Jumping Beans, Grasshopper, Swarm, and JAMES). The study provides some information about the performance and robustness of each platform. The experiment involved a cluster of six machines connected through a 10Mb/sec switched Ethernet. A simple benchmark application was used that was composed of a migratory agent that roamed the network to obtain a report about the current memory usage of each machine. The authors were investigating three main metrics, namely application performance; robustness; and network traffic. A summary of their findings is presented in [13].

[14] compares a FIPA-compliant agent technology with a non-agent J2EE component based architecture. It describes a comparison framework that reveals considerable differences in terms of performance, deployment support and reliability between the two technologies, and makes recommendations about when to select either technology.

7 Conclusions and Further Work

This paper has presented the results of evaluating three different agent technologies. The evaluation process has been effective in highlighting some of the strengths and weaknesses of each technology. As in any technology evaluation and selection process, there are few, if any, absolutes. The most important outcome is that the capabilities of a particular technology match the foreseeable application requirements.

We are now working on exploiting the outcomes of this project and extending its findings. This includes providing an effective means for disseminating the findings of this study and creating an on-going vehicle for evaluating additional agent technologies that researchers and developers may be interested in. Like [15], we believe that
more widespread use of rigorous quantitative evaluation will help more rapidly mature the spectrum of agent technologies. To this end, further evaluation studies incorporating different agent technologies, larger scale test implementations and heterogeneous distributed test-beds are desirable.

References

<table>
<thead>
<tr>
<th>Author</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bai, Xin</td>
<td>149</td>
</tr>
<tr>
<td>Berbers, Yolande</td>
<td>92</td>
</tr>
<tr>
<td>Blake, M. Brian</td>
<td>111</td>
</tr>
<tr>
<td>Bölöni, Ladislau</td>
<td>149</td>
</tr>
<tr>
<td>Breitman, Karin Koogan</td>
<td>19</td>
</tr>
<tr>
<td>Bresciani, Paolo</td>
<td>1, 35</td>
</tr>
<tr>
<td>Cabri, Giacomo</td>
<td>243</td>
</tr>
<tr>
<td>Campo, Marcelo</td>
<td>73</td>
</tr>
<tr>
<td>Chavez, Christina</td>
<td>49</td>
</tr>
<tr>
<td>Choren, Ricardo</td>
<td>129</td>
</tr>
<tr>
<td>Cowell, Andrew</td>
<td>264</td>
</tr>
<tr>
<td>Díaz Pace, Andrés</td>
<td>73</td>
</tr>
<tr>
<td>Dony, Christophe</td>
<td>167</td>
</tr>
<tr>
<td>Donzelli, Paolo</td>
<td>1</td>
</tr>
<tr>
<td>Fernandes, Elizabeth</td>
<td>207</td>
</tr>
<tr>
<td>Ferrari, Luca</td>
<td>243</td>
</tr>
<tr>
<td>Garcia, Alessandro</td>
<td>49</td>
</tr>
<tr>
<td>Giorgini, Paolo</td>
<td>35</td>
</tr>
<tr>
<td>Gomaa, Hassan</td>
<td>111</td>
</tr>
<tr>
<td>Gorton, Ian</td>
<td>264</td>
</tr>
<tr>
<td>Haack, Jereme</td>
<td>264</td>
</tr>
<tr>
<td>Holvoet, Tom</td>
<td>92</td>
</tr>
<tr>
<td>Horling, Bryan</td>
<td>225</td>
</tr>
<tr>
<td>Ji, Yongchang</td>
<td>149</td>
</tr>
<tr>
<td>Jung, Galeno A.</td>
<td>207</td>
</tr>
<tr>
<td>Khan, Majid Ali</td>
<td>149</td>
</tr>
<tr>
<td>Kuchar, Olga</td>
<td>264</td>
</tr>
<tr>
<td>Lesser, Victor</td>
<td>225</td>
</tr>
<tr>
<td>Lucena, Carlos J.P. de</td>
<td>49, 129</td>
</tr>
<tr>
<td>Mailler, Roger</td>
<td>225</td>
</tr>
<tr>
<td>Manson, Gordon</td>
<td>35</td>
</tr>
<tr>
<td>Marinescu, Dan C.</td>
<td>149</td>
</tr>
<tr>
<td>McGee, David</td>
<td>264</td>
</tr>
<tr>
<td>Minsky, Naftaly H.</td>
<td>189</td>
</tr>
<tr>
<td>Mouratidis, Haralambos</td>
<td>35</td>
</tr>
<tr>
<td>Murata, Takahiro</td>
<td>189</td>
</tr>
<tr>
<td>Obelheiro, Rafael R.</td>
<td>207</td>
</tr>
<tr>
<td>Saint Germain, Bart</td>
<td>92</td>
</tr>
<tr>
<td>Sampaio do Prado Leite, Julio Cesar</td>
<td>19</td>
</tr>
<tr>
<td>Sant’Anna, Cláudio</td>
<td>49</td>
</tr>
<tr>
<td>Schelfhout, Kurt</td>
<td>92</td>
</tr>
<tr>
<td>Silva, Viviane Torres da</td>
<td>49, 129</td>
</tr>
<tr>
<td>Silva Fraga, Joni da</td>
<td>207</td>
</tr>
<tr>
<td>Soria, Alvaro</td>
<td>73</td>
</tr>
<tr>
<td>Souchon, Frédéric</td>
<td>167</td>
</tr>
<tr>
<td>Staa, Arndt von</td>
<td>49</td>
</tr>
<tr>
<td>Steegmans, Elke</td>
<td>92</td>
</tr>
<tr>
<td>Thomson, Judi</td>
<td>264</td>
</tr>
<tr>
<td>Urtado, Christelle</td>
<td>167</td>
</tr>
<tr>
<td>Valckenaers, Paul</td>
<td>92</td>
</tr>
<tr>
<td>Vauttier, Sylvain</td>
<td>167</td>
</tr>
<tr>
<td>Wang, Guoqiang</td>
<td>149</td>
</tr>
<tr>
<td>Wangham, Michelle S.</td>
<td>207</td>
</tr>
<tr>
<td>Zambonelli, Franco</td>
<td>243</td>
</tr>
</tbody>
</table>