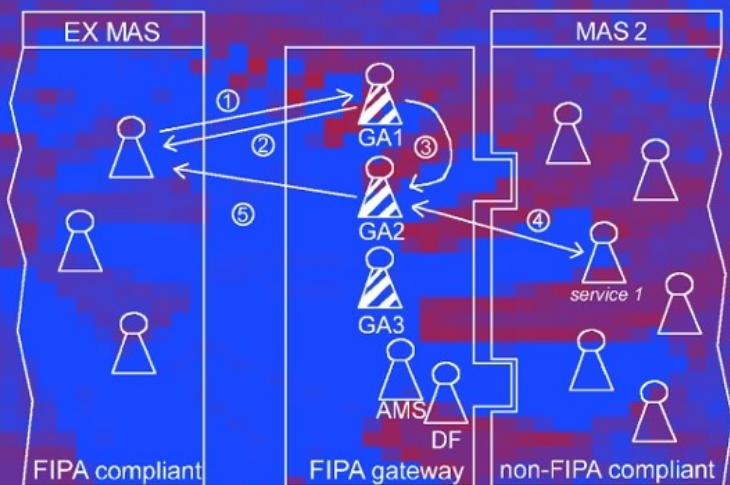


Paolo Giorgini
Jörg P. Müller
James Odell (Eds.)

Agent-Oriented Software Engineering IV

4th International Workshop, AOSE 2003
Melbourne, Australia, July 2003
Revised Papers



Springer

Lecture Notes in Computer Science
Edited by G. Goos, J. Hartmanis, and J. van Leeuwen

2935

Springer

Berlin

Heidelberg

New York

Hong Kong

London

Milan

Paris

Tokyo

المنارة للاستشارات

Paolo Giorgini Jörg P. Müller
James Odell (Eds.)

Agent-Oriented Software Engineering IV

4th International Workshop, AOSE 2003
Melbourne, Australia, July 15, 2003
Revised Papers



Springer

المشاركة للإصدارات

Series Editors

Gerhard Goos, Karlsruhe University, Germany
Juris Hartmanis, Cornell University, NY, USA
Jan van Leeuwen, Utrecht University, The Netherlands

Volume Editors

Paolo Giorgini
University of Trento, Department of Information and Communication Technology
Via Sommarive, 14, 38050 Povo, Trento, Italy
E-mail: paolo.giorgini@dit.unitn.it

Jörg P. Müller
Siemens AG, Corporate Technology
Intelligent Autonomous Systems
Otto-Hahn-Ring 6, 81730 Munich, Germany
E-mail: joerg.p.mueller@siemens.com

James Odell
James Odell Associates
3646 West Huron River Drive, Ann Arbor, MI 48103, USA
E-mail: email@jamesodell.com

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, F.3, D.1, D.2.4, D.3

ISSN 0302-9743
ISBN 3-540-20826-7 Springer-Verlag Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer-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 PTP-Berlin, Protago-TeX-Production GmbH
Printed on acid-free paper SPIN: 10981368 06/3142 5 4 3 2 1 0

Preface

The explosive growth of application areas such as electronic commerce, enterprise resource planning and mobile computing has profoundly and irreversibly changed our views on software systems. Nowadays, software is to be based on open architectures that continuously change and evolve to accommodate new components and meet new requirements. Software must also operate on different platforms, without recompilation, and with minimal assumptions about its operating environment and its users. Furthermore, software must be robust and autonomous, capable of serving a naive user with a minimum of overhead and interference.

Agent concepts hold great promise for responding to the new realities of software systems. They offer higher-level abstractions and mechanisms that address issues such as knowledge representation and reasoning, communication, coordination, cooperation among heterogeneous and autonomous parties, perception, commitments, goals, beliefs, and intentions, all of which need conceptual modeling. On the one hand, the concrete implementation of these concepts can lead to advanced functionalities, e.g., in inference-based query answering, transaction control, adaptive workflows, brokering and integration of disparate information sources, and automated communication processes. On the other hand, their rich representational capabilities allow more faithful and flexible treatments of complex organizational processes, leading to more effective requirements analysis and architectural/detailed design.

In keeping with its very successful predecessors, AOSE 2000, AOSE 2001, and AOSE 2002 (Lecture Notes in Computer Science Volumes 1957, 2222, and 2585), the AOSE 2003 workshop sought to examine the credentials of agent-based approaches as a software engineering paradigm, and to gain an insight into what agent-oriented software engineering will look like.

AOSE 2003 was hosted by the 2nd International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2003) held in Melbourne, Australia on July 2003. The workshop received 43 submissions, and 15 of them were accepted for presentation (an acceptance rate of 30%). These papers were reviewed by at least 3 members of an international program committee composed of 25 researchers. The submissions followed a call for papers on all aspects of agent-oriented software engineering, and showed the range of results achieved in several areas, such as methodologies, modeling, architectures, and tools.

The workshop program included an invited talk, a technical session in which the accepted papers were presented and discussed, and a closing plenary session. It congregated more than 50 attendees, among them researchers, students, and practitioners, who contributed to the discussion of research problems related to the main topics in AOSE.

This volume contains revised versions of the 15 papers presented at the workshop. Additionally, it contains an invited contribution by Bernhard Bauer and Jörg Müller on “Using UML in the Context of Agent-Oriented Software Engineering: State of the Art.” We believe that this thoroughly prepared volume

is of particular value to all readers interested in the key topics and most recent developments in the very exciting field of agent-oriented software engineering.

We thank the authors, the participants, and the reviewers for making AOSE 2003 a high-quality scientific event.

November 2003

Paolo Giorgini
Jörg P. Müller
James Odell

Organization

Organizing Committee

Paolo Giorgini (Co-chair)
Department of Information and Communication Technology
University of Trento, Italy
Email: paolo.giorgini@dit.unitn.it

Jörg P. Müller (Co-chair)
Siemens AG, Germany
Email: joerg.mueller@mchp.siemens.de

James Odell (Co-chair)
James Odell Associates, Ann Arbor, MI, USA
Email: email@jamesodell.com

Steering Committee

Paolo Ciancarini, University of Bologna, Italy
Gerhard Weiss, Technische Universitaet Muenchen, Germany
Michael Wooldridge, University of Liverpool, UK

Program Committee

Bernard Bauer (Germany)	Yannis Labrou (USA)
Federico Bergenti (Italy)	Juergen Lind (Germany)
Scott DeLoach (USA)	John Mylopolous (Canada)
Marie-Pierre Gervais (France)	Andrea Omicini (Italy)
Olivier Gutknecht (France)	Van Parunak (USA)
Brian Henderson-Sellers (Australia)	Anna Perini (Italy)
Michael Huhns (USA)	Marco Pistore (Italy)
Carlos Iglesias (Spain)	Onn Shehory (Israel)
Nicholas Jennings (UK)	Gerhard Weiss (Germany)
Catholijn Jonker (Netherlands)	Paola Turci (Italy)
Liz Kendall (Australia)	Eric Yu (Canada)
David Kinny (Australia)	Franco Zambonelli (Italy)
Manuel Kolp (Belgium)	

Auxiliary Reviewers: Paolo Busetta, Julio Cesar Leite, Aizhong Lin, Matthias Nickles, Michael Rovatsos, Marco Roveri, Arnon Sturm, Angelo Susi, Martijn Schut

Table of Contents

Modeling Agents and Multiagent Systems

Using UML in the Context of Agent-Oriented Software Engineering: State of the Art	1
<i>Bernhard Bauer, Jörg P. Müller</i>	
Towards a Recursive Agent Oriented Methodology for Large-Scale MAS	25
<i>Adriana Giret, Vicente Botti</i>	
Agent-Oriented Modeling by Interleaving Formal and Informal Specification	36
<i>Anna Perini, Marco Pistore, Marco Roveri, Angelo Susi</i>	
The ROADMAP Meta-model for Intelligent Adaptive Multi-agent Systems in Open Environments	53
<i>Thomas Juan, Leon Sterling</i>	
Modeling Deployment and Mobility Issues in Multiagent Systems Using AUML	69
<i>Agostino Poggi, Giorgio Rimassa, Paola Turci, James J. Odell, Haralabos Mouratidis, G. Manson</i>	

Methodologies and Tools

A Knowledge-Based Methodology for Designing Reliable Multi-agent Systems	85
<i>Mark Klein</i>	
A Framework for Constructing Multi-agent Applications and Training Intelligent Agents	96
<i>Pericles A. Mitkas, Dionisis Kehagias, Andreas L. Symeonidis, Ioannis N. Athanasiadis</i>	
Activity Theory for the Analysis and Design of Multi-agent Systems	110
<i>Rubén Fuentes, Jorge J. Gómez-Sanz, Juan Pavón</i>	
A Design Taxonomy of Multi-agent Interactions	123
<i>H. Van Dyke Parunak, Sven Brueckner, Mitch Fleischer, James J. Odell</i>	
Automatic Derivation of Agent Interaction Model from Generic Interaction Protocols	138
<i>José Ghislain Quenum, Aurélien Slodzian, Samir Aknine</i>	

Patterns, Architectures, and Reuse

Building Blocks for Agent Design	153
<i>Hrishikesh J. Goradia, José M. Vidal</i>	
Supporting FIPA Interoperability for Legacy Multi-agent Systems	167
<i>Christos Georgousopoulos, Omer F. Rana, Anthony Karageorgos</i>	
Dynamic Multi-agent Architecture Using Conversational Role Delegation	185
<i>Denis Jouwin, Salima Hassas</i>	

Roles and Organizations

Temporal Aspects of Dynamic Role Assignment	201
<i>James J. Odell, H. Van Dyke Parunak, Sven Brueckner, John Sauter</i>	
From Agents to Organizations: An Organizational View of Multi-agent Systems	214
<i>Jacques Ferber, Olivier Gutknecht, Fabien Michel</i>	
Modelling Multi-agent Systems with Soft Genes, Roles, and Agents	231
<i>Qi Yan, XinJun Mao, Hong Zhu, ZhiChang Qi</i>	
Author Index	247

Using UML in the Context of Agent-Oriented Software Engineering: State of the Art

Bernhard Bauer¹ and Jörg P. Müller²

¹ Institute of Computer Science, University of Augsburg, D-86135 Augsburg, Germany
Bernhard.Bauer@informatik.uni-augsburg.de

² Siemens AG, Corporate Technology, CT IC 6, D-81730 Munich, Germany
joerg.p.mueller@siemens.com

Abstract. Most of the methodologies and notations for agent-oriented software engineering developed over the past few years are based on the Unified Modeling Language (UML) or proposed extensions of UML. However, at the moment an overview on the different approaches is missing. In this paper, we present a state-of-the-art survey of the different methodologies and notations that, in one way or the other, rely on the usage of UML for the specification of agent-based systems. We focus on two aspects, i.e., design methodologies for agent-oriented software engineering, and different types of notations (e.g., for interaction protocols, social structures, or ontologies) that rely on UML.¹

1 Introduction

The complexity of commercial software development processes increasingly requires the usage of software engineering techniques, including methodologies and tools for building, deploying, and maintaining software systems and solutions. In this context, software methodologies play a key role. A *software methodology* is typically characterized by a *modeling language* – used for the description of models, defining the elements of the model together with a specific syntax (notation) and associated semantics – and a *software process* – defining the development activities, the interrelationships among the activities, and how the different activities are performed. In particular, the software process defines phases for process and project management as well as quality assurance. The three key phases that one is likely to find in any software engineering process are that of analysis, design and implementation. In a strict waterfall model these are the only phases; more recent software development process models employ a “round trip engineering” approach, i.e., provide an iteration of smaller granularity cycles, in which models developed in earlier phases can be refined and adapted in later phases.

Agent technology enables the realization of complex software systems characterized by situation awareness and intelligent behavior, a high degree of distribution, as well as mobility support. Over the past year, agents have been very successful from the scientific point of view; also, the beginning commercial success of agent technology at the application level (in the sense of: intelligent components

¹ This paper is a short and adapted version of [42]

supporting intelligent applications, see e.g., [44]) is evident today. However, the potential role of agent technology as a new paradigm for software engineering has not yet met with broad acceptance in industrial and commercial settings. We claim that the main reason for this is the lack of accepted methods for software development depending on widely standardized representations of artifacts supporting all phases of the software lifecycle. In particular, these standardized representations are needed by tool developers to provide commercial quality tools that mainstream software engineering departments need for industrial agent systems development.

Currently, most industrial methodologies are based on the Object Management Group's (OMG) Unified Modeling Language (UML) accompanied by process frameworks such as the Rational Unified Process (RUP), see [28] for details. The Model-Driven Architecture (MDA [40]) from the OMG allows a cascade of code generations from high-level models (platform independent model) via platform dependent models to directly executable code (e.g., see the tool offered by Kennedy Carter [39]).

Thus, one possibility to provide an answer regarding the state-of-the-art in agent-oriented software engineering is to look at the level of support currently provided for UML technologies by recent agent-based engineering approaches. In this paper we will provide a detailed survey of methodologies and notations for agent-based engineering of software systems based on UML.

In Section 2 we will have a closer look at different methodologies for designing agent-based systems. In Section 3 focuses on notations based on UML. In particular, we shall look at notations for interaction protocols, social structures, agent classes, ontologies, and goals and plans. The paper concludes with a summary and an outlook for further research in Section 4.

2 Methodologies

In this we will take a closer look at agent methodologies that directly extend object-oriented – UML approaches. In the next section we will also give an overview of UML notations and extensions available for the specification of agent-based systems. Since most of the notations use graphical representations of software artifacts we will use examples taken from the original research papers.

2.1 Agent Modeling Techniques for Systems of BDI Agents

One of the first methodologies for the development of BDI agents based on OO technologies was presented in [2][3][4][5]. The agent methodology distinguishes between the *external viewpoint* - the system is decomposed into agents, modeled as complex objects characterized by their purpose, their responsibilities, the services they perform, the information they require and maintain, and their external interactions - and the *internal viewpoint* - the elements required by a particular agent architecture must be modeled for each agent, i.e. an agent's beliefs, goals, and plans. For each of these views different models are described (based on [2] and [5]):

The *external view* is characterized by two models which are largely independent of the underlying BDI architecture:

Agent Model: This model describes the hierarchical relationship among different abstract and concrete agent classes (*Agent Class Model*) similar to a UML class diagram denoting both abstract and concrete (instantiable) agent classes, inheritance and aggregation as well as predefined reserved attributes, e.g., each class may have associated belief, goal, and plan models; and identifies the agent instances which may exist within the system, their multiplicity, and when they come into existence (*Agent Instance Model*) with the possibility to define *initial-belief-state* and *initial-goal-state* attributes.

Interaction Model: describes the responsibilities of an agent class, the services it provides, associated interactions, and control relationships between agent classes. This includes the syntax and semantics of messages used for inter-agent communication and communication between agents and other system components, such as user interfaces.

BDI agents are *internally viewed* as having certain mental attitudes, *Beliefs*, *Desires* and *Intentions*, which represent, respectively, their informational, motivational and deliberative states. These aspects are captured, for each agent class, by the following models.

Belief Model describes the information about the environment and internal state that an agent of that class may hold, and the actions it may perform. The possible beliefs of an agent and their properties, such as whether or not they may change over time, are described by a *belief set*. In addition, one or more *belief states* - particular instances of the belief set - may be defined and used to specify an agent's initial mental state. The *belief set* is specified by a set of object diagrams which define the domain of the beliefs of an agent class. A belief state is a set of instance diagrams which define a particular instance of the belief set. Formally, defined by a set of typed predicates whose arguments are terms over a universe of predefined and user-defined function symbols.

Goal Model describes the goals that an agent may possibly adopt, and the events to which it can respond. It consists of a *goal set* which specifies the goal and event domain and one or more *goal states* - sets of ground goals - used to specify an agent's initial mental state. A goal set is, formally, a set of goal formula signatures. Each such formula consists of a modal goal operator applied to a predicate from the belief set.

Plan Model describes the plans that an agent may possibly employ to achieve its goals. It consists of a *plan set* which describes the properties and control structure of *individual plans*. Plans are modeled similar to simple UML State Chart Diagrams, which can be directly executed showing how an agent should behave to achieve a goal or respond to an event. In contrast to UML activities may be sub-goals, denoted by formulae from the agent's goal set; conditions are predicates from the agent's belief set; actions include those defined in the belief set, and built-in actions. The latter include assert and retract, which update the belief state of the agent.

2.2 Message

MESSAGE (Methodology for Engineering Systems of Software Agents) [6][7] is a methodology which builds upon best practice methods in current software engineering such as for instance UML for the analysis and design of agent-based systems. It consists of (i) applicability guidelines; (ii) a modeling notation that

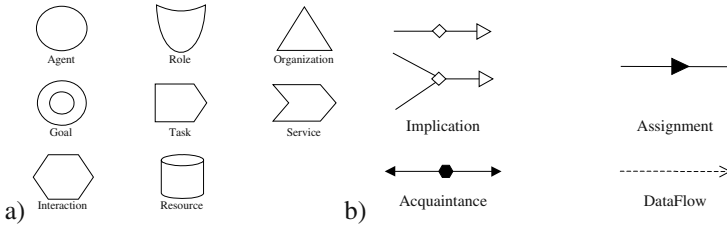


Fig. 1. a) concept symbols in MESSAGE; b) relations in MESSAGE

extends UML by agent-related concepts (inspired e.g. by Gaia); and (iii) a process for analysis and design of agent systems based on Rational unified Process. The MESSAGE modeling notation extends UML notation by key agent-related concepts. We describe the notation used in MESSAGE based on the example presented in [7]. For details on the example we refer to this paper. The used concept and relation symbols are shown in Fig. 1.

The main focus of MESSAGE is on the phase of analysis of agent-based systems. For this purpose, MESSAGE presents five analysis models, which analysts can use to capture different aspects of an agent-based system. The models are described in terms of sets of interrelated concepts. The five models are (following [7][6]):

Organization Model: The Organization Model captures the overall structure and the behavior of a group of agents and the external organization working together to reach common goals. In particular, it represents the responsibilities and authorities with respect to entities such as processes, information, and resources and the structure of the organization in terms of sub-organization such as departments, divisions, sections, etc. expressed through power relationships (e.g. superior-subordinate relationships). Moreover it provides the *social view* characterizing the overall behavior of the group, whereas the agent model covers the *individual view* dealing with the behavior of agents to achieve common/social goals. It offers software designers a useful abstraction for understanding the overall structure of the multi-agent system, what the agents are, what resources are involved, what the role of each agent is, what their responsibilities are, which tasks are achieved individually and which achieved through co-operation. Different types of organization diagrams are available in MESSAGE to support the graphical representation of social concepts (see Fig. 2).

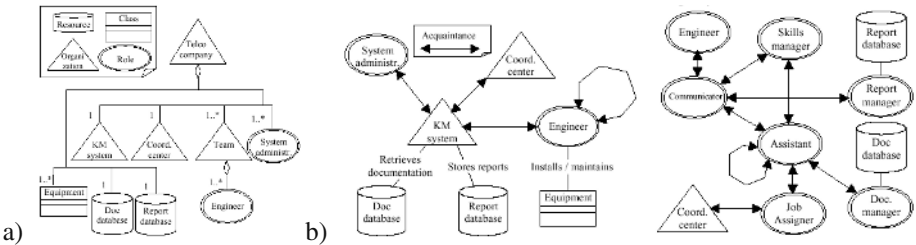


Fig. 2. Examples of organization diagrams: a) structural relationships, b) acquaintance relationships (analysis phase 0 and 1)

Goal/Task Model: The Goal/Task Model defines the goals of the composite system, i.e. the agent system and its environment, and their decomposition into sub-goals; the responsibility of agents for their commitments; the performance of tasks and actions by agents, the goals the tasks satisfy and the decomposition of tasks into sub-tasks as well as to describe tasks involved in an organizational workflow. It captures what the agent system and constituent agents do in terms of the goals that they work to attain and the tasks they must accomplish. The model also captures the way that goals and tasks of the system as a whole are related to goals and tasks assigned to specific agents and the dependencies among them. Goals and tasks both have attributes of type Situation, such that they can be linked by logical dependencies to form graphs that show e.g. decomposition of high-level goals into sub-goals, and how tasks can be performed to achieve goals. UML Activity Diagrams are applied for presentation purposes. Goals describe the desired states of the system and its environment, whereas tasks describe state transitions that are needed to satisfy agent goal commitments. The state transition is specified as a pre-and post-condition attribute pair. Actions are atomic tasks that can be performed by the agents to satisfy their goal commitments. Task inputs are Model Elements (adapted from UML defining elements composing models) that are processed in task. Task outputs are updates of the input Model Elements plus any new Model Element produced by the task. The desired states of a Model Element are specified by attributes called invariants, which are conditions that should always be true. An example is shown in Fig. 3.

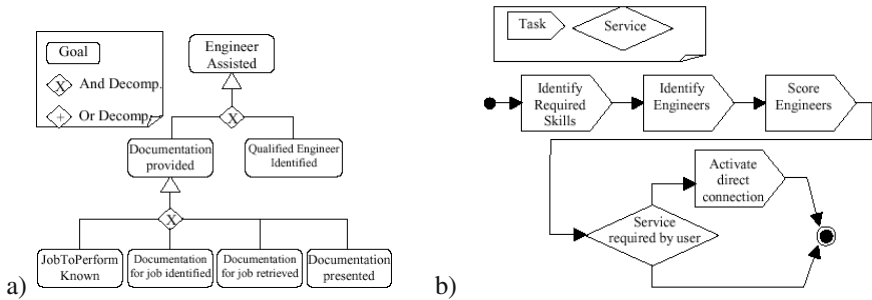


Fig. 3. Example of a) goal implication diagram, b) workflow diagram

Agent/Role Model: The Agent Model consists of a set of individual agents and roles. The relationship between role and agent is defined analogous to that between an interface and an object class: a role describes the external characteristics of an agent in a particular context. An agent may be capable of playing several roles, and multiple agents may be able to play the same role. roles can also be used as indirect references to agents. An element of the Agent Model gathers together information specific to an individual agent or role, including its relationships to other entities. In particular, it contains a detailed and comprehensive description of each individual agent providing an internal view including the agent's goals and the services, i.e. the functional capability, they provide. In contrast to the external perspective provided by the Organization Model. For each agent/role it uses schemata supported by diagrams to

define its characteristics such as what goals it is responsible for, what events it needs to sense, what resources it controls, what tasks it knows how to perform, 'behavior rules', etc. An example for an agent model is given in Fig. 4.

The Domain (Information) Model: The Domain Model functions as a repository of relevant information about the problem domain. The conceptualization of the specific domain is assumed to be a mixture of *object-oriented*, i.e. all entities in the domain are classified in classes and each class groups all entities with a common structure, and *relational*, i.e. a number of relations describe the mutual relationships between the entities belonging to the different classes. Thus the Domain Model defines the *domain-specific classes* agents deal with and describes the structure of each class in terms of a number (possibly null) of attributes having values that can belong to primitive types or can be instances of other domain specific classes.

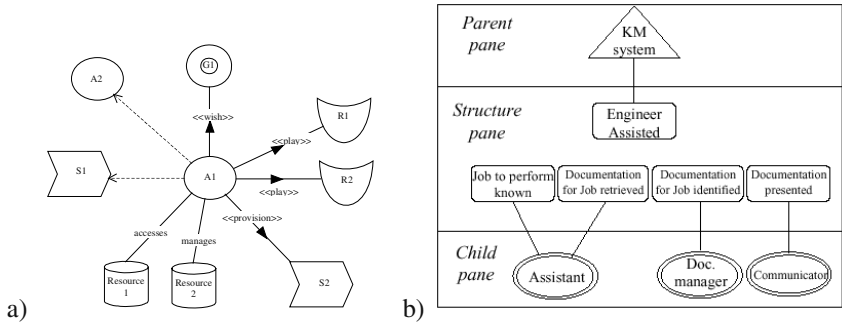


Fig. 4. Example of a) agent diagram², b) delegation structure diagram

In addition, *domain specific relations* holding among the instances of the domain specific classes are captured. Class diagrams are used for this model, as illustrated:

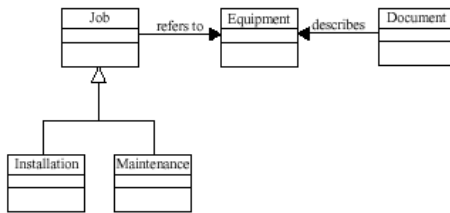


Fig. 5. Example of Domain Model as UML class diagrams

The Interaction Model: The Interaction Model is concerned with capturing the way in which agents (or roles) exchange information with one another (as well as with their environment captures). The content of the messages within an interaction may be described in the Domain Model. Interactions are specified from both a high-level and low-level perspective (interaction protocols based on the UML interaction protocols).

² Taken from [6].

For each interaction among agents/roles, shows the initiator, the collaborators, the motivator (generally a goal the initiator is responsible for), the relevant information supplied/achieved by each participant, the events that trigger the interaction, other relevant effects of the interaction (e.g. an agent becomes responsible for a new goal). Larger chains of interaction across the system (e.g. corresponding to uses cases) can also be considered such as delegation or workflows. An example for interaction is shown in Fig. 6 and on agent interaction diagrams.

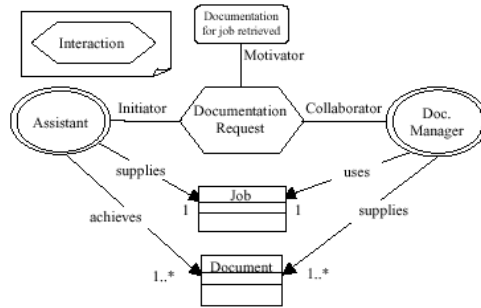


Fig. 6. Example of an Interaction

2.3 Tropos

Tropos [27][30][29] is another good example of a agent-oriented software development methodology that is based on object-oriented techniques. In particular, Tropos relies on UML and offers processes for the application of UML mainly for the development of BDI agents and the agent platform JACK [34]. Some elements of UML (like class, sequence, activity and interaction diagrams) are adopted as well for modeling object and process perspectives. The concepts of i^* [32] such as actor (actors can be agents, positions or roles), as well as social dependencies among actors (including goal, soft goal, task and resource dependencies) are embedded in a modeling framework which also supports generalization, aggregation, classification, and the notion of contexts [33]. Thus, Tropos was developed around two key features: Firstly, the notions of agent, goal, plan and various other knowledge-level concepts are provided as fundamental primitives used uniformly throughout the software development process; secondly, a crucial role is assigned to requirements analysis and specification when the system-to-be is analyzed with respect to its intended environment using a phase model: *Early Requirements*: identify relevant stakeholders (represented as actors), along with their respective objectives (represented as goals); *Late Requirements*: introduce system to be developed as an actor describing the dependencies to other actors indicating the obligations of the system towards its environment; *Architectural Design*: introduce more system actors assigned sub-goals or subtasks of the goals and tasks assigned to the system;

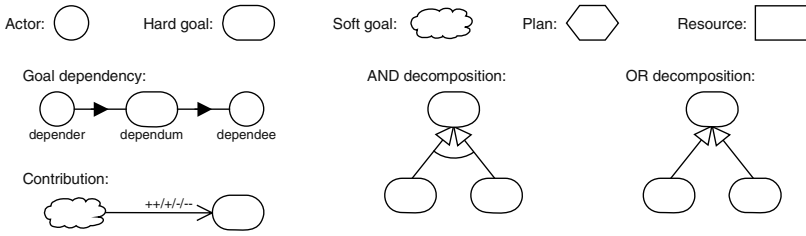


Fig. 7. Examples of Tropos notation

Detailed Design: define system actors in detail, including communication and coordination protocols; *Implementation:* transform specifications into a skeleton for the implementation mapping from the Tropos constructs to those of an agent programming platform. The specification covers the following notation illustrated in Fig. 7.

The Tropos specification makes use of the following types of models (following [27]):

Actor and Dependency Model: Actor and dependency models graphically represented through actor diagrams result from the analysis of social and system actors, as well as of their goals and dependencies for goal achievement as shown in Fig. 8. An actor has strategic goals and intentionality and represents a physical agent (e.g., a person), or a software agent as well as a role (abstract characterization of the behavior of an actor within some specialized context) or a position (a set of roles, typically played by one agent). An agent can occupy a position, while a position is said to cover a role. Actor models are extended during the late requirements phase by adding the system as another actor, along with its inter-dependencies with social actors. Actor models at the architectural design level provide a more detailed account of the system-to-be actor and its internal structure. This structure is specified in terms of subsystem actors, interconnected through data and control flows that are modeled as dependencies. A dependency between two actors indicates that one actor depends on another in order to attain some goal, execute some plan, or deliver a resource. By depending on other actors, an actor is able to achieve goals that it would otherwise be unable to achieve on its own, or not as easily, or not as well.

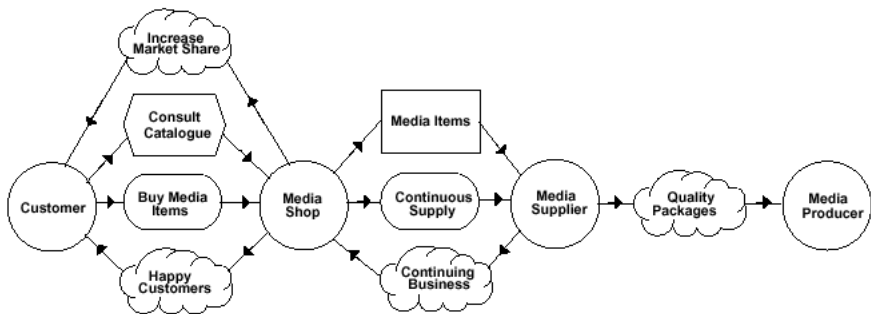


Fig. 8. Actor Diagram in Tropos (taken from [33])

Goal and Plan models: Goal and plan models allow the designer to analyze goals representing the strategic interests of actors and plans representing a way of a goal is satisfied from the perspective of a specific actor by using three basic reasoning techniques: *means-end analysis* refining a goal into subgoals in order to identify plans, resources and soft goals that provide means for achieving the goal (the end); *contribution analysis* pointing out goals that can contribute positively or negatively in reaching the goal being analyzed, and *AND/OR decomposition* allowing to combination of AND and OR decompositions of a root goal into sub-goals, thereby refining a goal structure. Between two kinds of goals is distinguished, namely hard goals and soft goals, the latter having no clear-cut definition and/or criteria as to whether they are satisfied. Goal models are first developed during early requirements using initially-identified actors and their goals.

Capability diagram: A capability, modeled either textually (e.g. as a list of capabilities for each actor) or as capability diagrams using UML activity from an agent's point of view, represents the ability of an actor to define, choose and execute a plan to fulfill a goal, given a particular operating environment. Starting states of a capability diagram are external events, whereas activity nodes model plans, transitions model events, and beliefs are modeled as objects. Each plan node of a capability diagram can be refined by UML activity diagrams.

Agent interaction diagrams: Protocols are modeled using the Agent UML sequence diagrams [1]

2.4 Prometheus

Similar to Tropos, Prometheus [37][36][35] is an iterative methodology covering the complete software engineering process and aiming at the development of intelligent agents using goals, beliefs, plans, and events, i.e. in particular BDI agents, resulting in a specification which can be implemented with JACK [34]. The Prometheus methodology covers three phases, namely those of System specification, architectural design, and detailed design. Fig. 9 illustrates the Prometheus process [35].

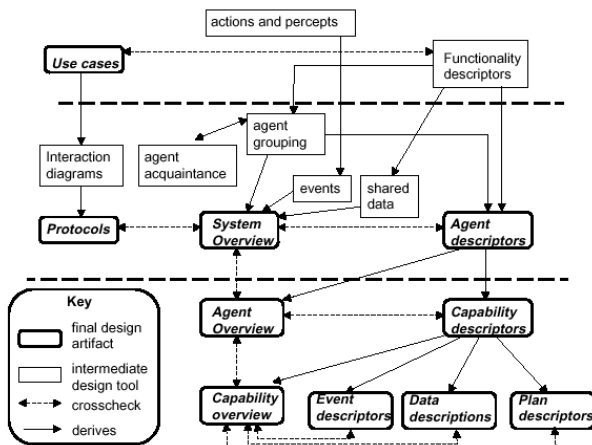


Fig. 9. Prometheus process overview

In the following, we describe the three phases of the Prometheus methodology according to [36], [35].

System Specification: The System Specifications focuses on identifying the basic functions of the system, along with inputs (percepts), outputs (actions) as well as their processing (e.g. how are percepts to be handled and any important shared data sources to model the system’s interaction with respect to its changing and dynamic environment. To understand the purpose of a system, use case scenarios borrowed from object-orientation with a slightly enhanced structure give a more holistic view than the mere analysis of the system functions in isolation.

Architectural Design: The architectural design phase subsequent to system specification determines which agents the system will contain and how they will interact. The major decision to be made during the architectural design is which agents should exist within the system. The key design artifacts used in this phase are the *system overview diagram* tying together agents, events and shared data objects, *agent descriptions* and the *interaction protocols* (based on Agent UML sequence diagrams [1]) specifying fully the interaction between agents. Agent messages are also identified, forming the interface between agents. Data objects are specified using traditional object oriented techniques. Taken the examples from [35] the diagrams look as illustrated in Fig. 10:

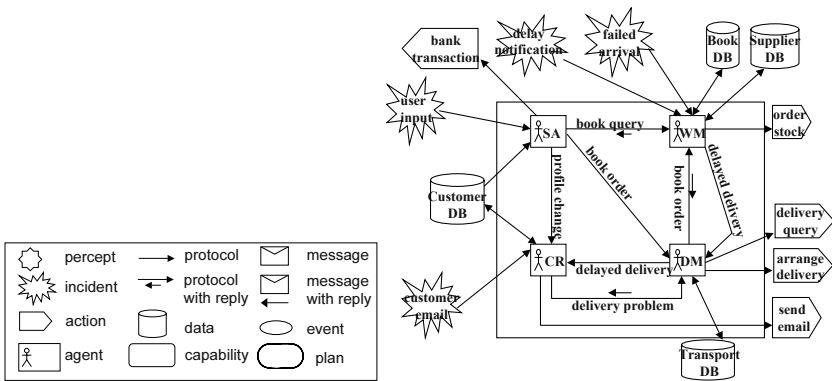


Fig. 10. Example of system overview diagram

Detailed design: The detailed design phase describes the internals of each agent and how it will achieve its tasks within the overall system. The focus is on defining capabilities (modules within the agent), internal events, plans and detailed data structures. Outcomes from this phase are *agent overview diagrams* (see Fig. 11a) providing the agent’s top-level capabilities, *capability diagrams* (see Fig. 11b), detailed *plan descriptors* and *data descriptions*. Capabilities can be nested within other capabilities; thus this model supports arbitrarily many layers in the detailed design, in order to achieve an understandable complexity at each level. They are refined until all capabilities are defined in terms of other capabilities, or (eventually) in terms of events, data, and plans.



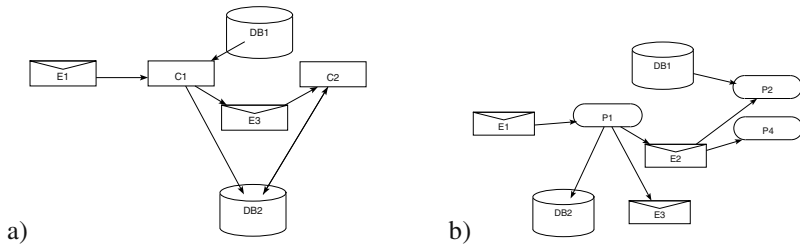


Fig. 11. Example of a) agent overview diagram, and b) capability overview diagram taken from [35]

2.5 MaSE

Multiagent Systems Engineering (MaSE) (we base our presentation on [24], for details we refer to [25][26]) has been developed to support the complete software development lifecycle from problem description to realization. It offers an environment for analyzing, designing, and developing heterogeneous multi-agent systems independent of any particular multi-agent system architecture, agent architecture, programming language, or message-passing system. It takes an initial system specification, and produces a set of formal design documents in a graphical style. In particular, MaSE offers the ability to track changes throughout the different phases of the process. The MaSE methodology is heavily based on UML and the RUP. The software development process is detailed in analysis and design. The different models to be covered are:

Capturing Goals: In this phase, the initial requirements are transformed into a structured set of system goals. A goal is always defined as a system-level objective. Goals are identified by distilling the essence of the set of requirements and are then analyzed and structured into a form that can be passed on and used in the design phases. Therefore the goals are organized by importance in a goal hierarchy diagram. Each level of the hierarchy contains goals that are roughly equal in scope and all sub-goals relate functionally to their parent.

Applying Use Cases: Use cases are drawn from the system requirements as in any UML analysis. Subsequently, sequence diagrams are applied to determine the minimum set of messages that must be passed between roles. Typically, at least one sequence diagram is derived from a use case.

Refining Roles: The roles and concurrent tasks are assigned from the goal hierarchy diagram and the sequence diagrams. A role in MaSE is an abstract description of an entity's expected function and encapsulates the system goals the entity is responsible for. MaSE allows a traditional role model and a methodology-specific role model including information on interactions between role tasks shown in Fig. 12.

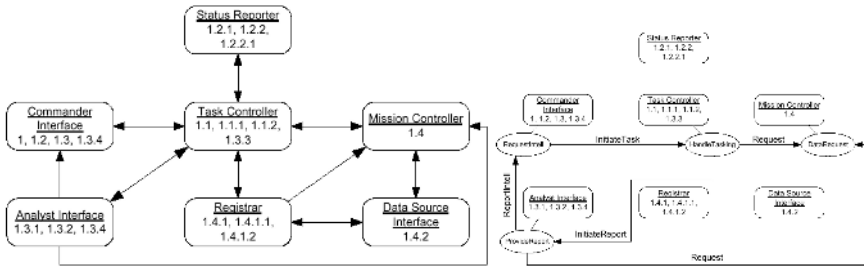


Fig. 12. MaSE a) traditional role model and b) MaSE role model.

Creating Agent Classes: The agent classes are identified from component roles. The result of this phase is an agent class diagram depicting agent classes and the conversations between them.

Constructing Conversations: A MaSE conversation defines a coordination protocol between two agents. Specifically, a conversation consists of two communication class diagrams, one each for the initiator and responder. A communication class diagram is a pair of finite state machines that define the conversation states of the two participant agent classes.

Assembling Agent Classes: the internals of agent classes are created based on the underlying architecture of the agents, like BDI, re-active agents, etc.

System Design: This model takes the agent classes and instantiates them as actual agents. It uses a Deployment Diagram to show the numbers, types, and locations of agents within a system.

2.6 PASSI

PASSI (Process for Agent Societies Specification and Implementation) [16][17] is an agent-oriented iterative requirement-to-code methodology for the design of multi-agent systems mainly driven from experiments in robotics. The methodology integrates design models and concepts from both object oriented software engineering and artificial intelligence approaches. PASSI is supported by a Rational Rose plug-in to have a dedicated design environment. In particular, automatic code generation for the models is partly supported and a focus lies on patterns and code reuse. We base our survey on [17].

The PASSI methodology consists of five models (System Requirements, Agent Society, Agent Implementation, Code Model and Deployment Model) which include several distinct phases as described in the following.

System Requirements Model: The System Requirements model is obtained in different phases: The *Domain Description* Phase results in a set of use case diagrams where scenarios are detailed using sequence diagrams. The next phase, namely the *Agent Identification*, defines, based on use cases, packages where the functionality of each agent is grouped and activity diagrams for the task specification of this agent. I.e., in contrast to most of the agent-oriented methodologies agents are identified based on their functionality and not on their roles. The *Role Identification* Phase is a functional/behavior description of the agents as well as a representation of its relationships to other agents described by a set of sequence diagrams. Roles are

viewed as in traditional object-oriented approaches. One activity diagram is drawn for each agent in the *Task Specification* Phase where each diagram is divided into two segments, one dealing with the tasks of an agents and one with the tasks for the interacting agent.

Agent Society Model: The agent society model is derived in the phases: *Ontology Description* describes the agent society or organization from an ontological point of view. Therefore two diagrams are introduced, the *Domain Ontology Description* and *Communication Ontology Description* usually presented using Class Diagrams and XML Schema for textual representation. The *Role Description* Phase models the life of the agents looking at its roles, therefore social or organizational roles and behavioral roles, represented by class diagrams where roles are classes grouped in packages representing the agents. In particular role changes can be defined. Roles are obtained by composing several tasks (roles are based on the functionality of an agent!). A part of such a diagram is shown in Fig. 13.

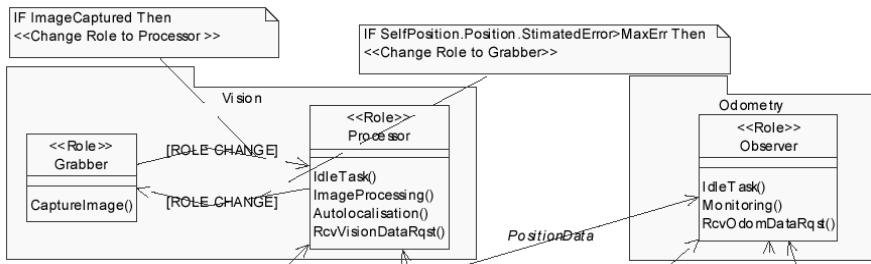


Fig. 13. Excerpt from PASSI role diagram

Agent Implementation Model: This model covers the *Agents Structure Definition* and the *Agents Behavior Description* Phases, describing respectively the *multi-agent level* represented by classes where attributes are the knowledge of the agent, methods are the tasks of an agent and relationships between agents define the communication between them; and the *single-agent level* defines one single class diagram for each agent, describing the complete structure of an agent with its attributes and methods. In particular, the methods needed to register the agent and for each task of the agent is represented as a class.

Code Model: Based on the FIPA standard architecture standard code pieces are available for re-use and therefore automatic code generation from the models is partly supported.

Deployment Model: UML deployment diagrams are extended to define the deployment of the agents and in particular to specify the behavior of mobile agents.

3 Modeling Notations Based on UML

The UML modeling notation is applied in various papers for the modeling of different aspects of agent-based software systems. While some approaches (e.g., [9], [10]) use plain UML 1.4 as a base notation for agent-based software development, there is a shared understanding, that UML as presented in version 1.4 is not sufficient for modeling agent-based systems [11]. The upcoming UML 2 standard will address

some current limitations and some parts of UML extension for agent-based systems will be taken into consideration in UML 2. Therefore in the following we will only present those extensions of UML 1.4, for which to our knowledge no updated version based on UML 2 is available.

3.1 Interaction Protocols

One of the first extensions to UML, in particular sequence diagrams were proposed in [12][1]. This notation was also applied as a basis for the specification of FIPA interaction protocols. In the meantime, this description was adapted to UML 2. An agent interaction protocol [13] is then represented as a sequence diagram as shown in Fig. 14 (taken from that source):

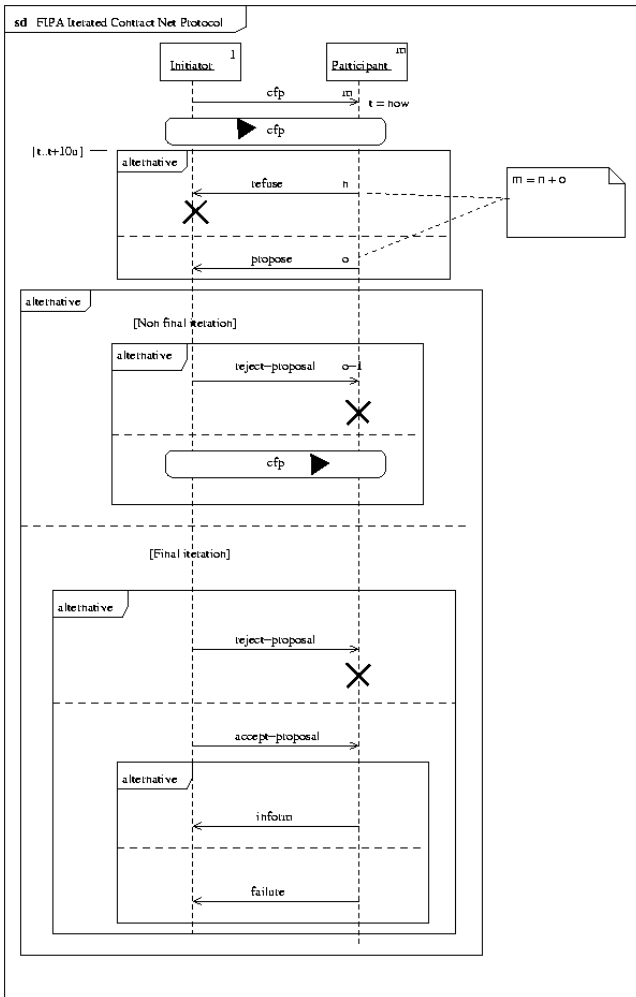


Fig. 14. FIPA Iterated Contract Net Protocol

In the contract net protocol, one agent takes the role of manager, e.g. a customer. The manager wishes to have some task performed by one or more other agents e.g. order some items, and further wishes to optimize a function that characterizes the task e.g. price and time of good. The Customer solicits proposals from the order acquisition by issuing a call for proposals (cfp), which specifies the task and any conditions the manager (Customer) is placing upon the execution of the order. Agents receiving the call for proposals are viewed as potential contractors, and are able to generate proposals to perform the task, e.g. the ordering as propose acts. The contractor's proposal e.g. Order Acquisition includes the preconditions that the contractor is setting out for the task, being the price and time when the order will be done. Alternatively, the contractor may refuse to propose or may iterate the process by issuing a revised cfp. The intent is that the Customer seeks to get better bids from the Order Acquisition by modifying the call and requesting new (equivalently, revised) bids. Once the Customer receives back replies from all of the Order Acquisition, it evaluates the proposals and makes its choice of which agents will perform the task. The process terminates when the Customer refuses all proposals and does not issue a new call, accepts one or more of the bids, or the Order Acquisitions all refuse to bid. The agents of the selected proposal(s) will be sent an acceptance message, the others will receive a notice of rejection. The proposals are assumed to be binding on the Order Acquisition, so that once the Customer accepts the proposal the Order Acquisition acquires a commitment to perform the task. Once the Order Acquisition has completed the task, it sends a completion message to the Customer.

3.2 Social Structures

Based on the emphasis on the correspondence between multi-agent systems and social systems, Parunak and Odell [38] combine several organizational models for agents, including AALAADIN, dependency theory, interaction protocols, and holonic modeling, in a general theoretical framework, and show how UML can be applied and extended to capture constructions in that framework. Parunak and Odell's model is based on the following artifacts: *roles*: They assume, that the same role can appear in multiple groups, if they embody the same pattern of dependencies and interactions. If an agent in a group holds multiple roles concurrently, it may sometimes be useful to define a higher-level role that is composed of some of those more elementary roles; *environments* environment are not only passive communications framework and everything of interest is relegated to it, but actively provides three information processing functions; It *fuses* information from different agents passing over the same location at different times; it *distributes* information from one location to nearby locations; it provides *truth maintenance* by forgetting information that is not continually refreshed, thereby getting rid of obsolete information; *Groups*: groups represent social units that are sets of agents associated by a common interest, purpose, or task. Groups can be created for three different reasons, i.e.: (i) for achieving more efficient or secure interaction between a set of agents (*intra-group associations*); (ii) for taking advantage between the synergies between a set of agents, resulting in an entity (the group) that is able to realize products, services, or processes that no individual by itself would be capable of (*group synergies*); and (iii) establishing a

group of agents that interacts with other agents or groups in a coherent way, e.g., to represent a shared position on a subject (*inter-group associations*).

The conceptual model of Parunak and Odell’s approach is illustrated in Fig. 15. In [38], the authors provide some examples for modeling social agent environments, namely a terrorist organization and its relationship to a weapons cartel. Groups are modeled by class diagrams and swimlanes as shown in Fig. 16, denoting that the Terrorist Organization involves two roles, Operative and Ringleader, where the Ringleader agent coordinates Operative agents.

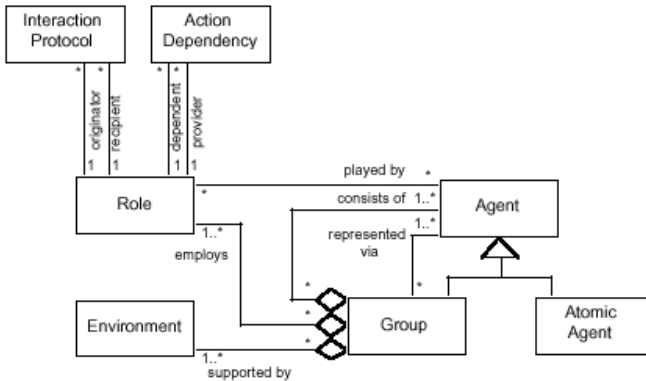


Fig. 15. Conceptual model of Parunak and Odell’s approach

The second swimlane is based on agent instances, e.g. agent A plays the roles of Operative, Customer, and Student (expressed in b).

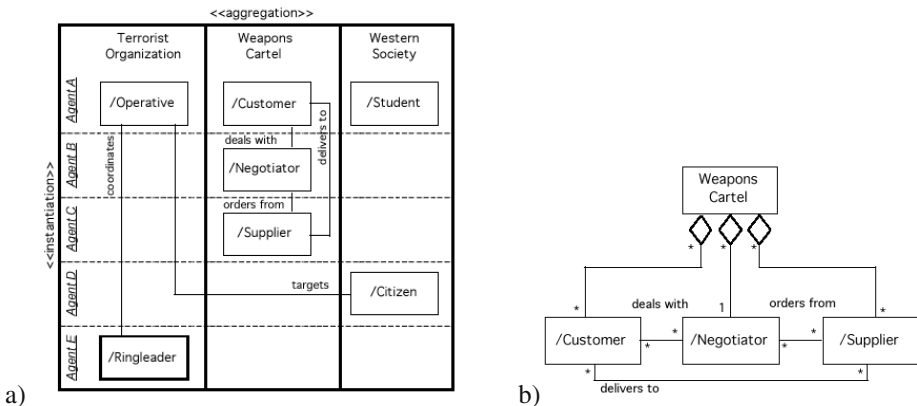


Fig. 16. a) Swimlanes as groups; b) class diagrams defines roles

Sequence diagrams are used to show roles as patterns of interactions; class diagrams model the kinds of entities that exist in a system along with their relationships, whereas sequence diagrams model the interactions that may occur among these entities. Fig. 17a) depicts the permitted interactions that may occur among Customer, Negotiator, and Supplier agents for a weapons procurement

negotiation. Fig. 17b) shows an activity graph modeling groups of agents as agents. In this way, the kinds of dependencies are expressed that are best represented at a group level.

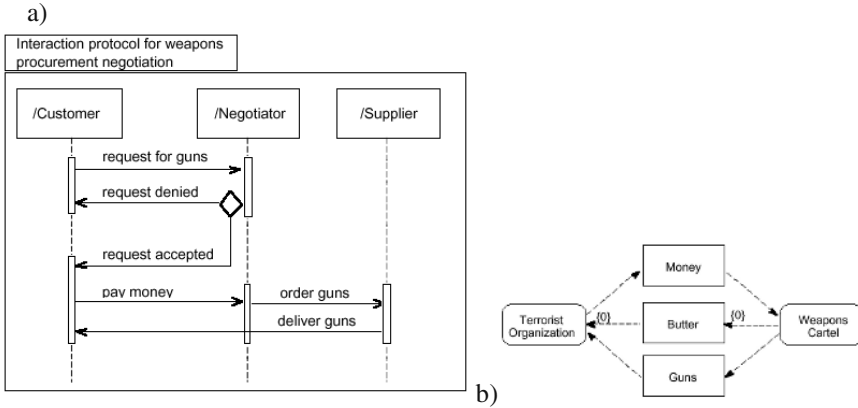


Fig. 17. a) Sequence diagram depicting an interaction protocol b) object-flow activity graph specifies roles as patterns of activities

3.3 Agent Classes

To our knowledge, basing agent classes on UML class diagrams was so far only considered by [15] and [14], with the notable exception of [43], where Wagner presents an UML profile for an agent-oriented modeling approach called an Agent-Object-Relationship modeling language (AORML). AORML can be viewed as an extension of UML covering (among others) *Interaction Frame Diagrams* describing the action event classes and commitment/claim classes determining the possible interactions between two agent types (or instances), *Interaction Sequence Diagrams* depicting prototypical instances of interaction processes, and *Interaction Pattern Diagrams* for representing general interaction patterns. The latter [14] is currently revisited within FIPA; an adapted version will be available by the end of 2003. Following [14] a distinction is made between an *agent class*, defining a blueprint for and the type of an individual agent, and between *individual agents* (being instances of an agent class). An agent class diagram shown in Fig. 18 specifies agent classes.

[14] states that usual UML notation with stereotypes can be used to define such an agent class, but for readability reasons the above notation was introduced:

Agent Class Descriptions and Roles: As we have seen agents can satisfy distinguished roles in most of the methodologies. The general form of describing agent roles in Agent UML [12] is

instance-1 ... instance-n / role-1 ... role-m : class

denoting a distinguished set of agent instances instance-1,..., instance-n satisfying the agent roles role-1,..., role-m with $n, m \geq 0$ and class it belongs to. Instances, roles or class can be omitted, for classes the role description is not underlined.

State description: A *state description* is similar to a field description in class diagrams with the difference that a distinguished class *wff* for *well-formed formula* for all kinds of logical descriptions of the state is introduced, independent of the

underlying logic. This extension allows the definition of e.g. BDI agents. Beyond the extension of the type for the fields, visibility and a persistency attributes can be added (denoted by the stereotype <<persistent>>) to allow the user agent to be stopped and re-started later in a new session. Optionally the fields can be initialized with some values. In the case of BDI semantics three instance variables can be defined, named *beliefs*, *desires*, and *intentions* of type *wff*. Describing the beliefs, desires, and intentions of a BDI agent. These fields can be initialized with the initial state of a BDI agent. The semantics state that the *wff* holds for the beliefs, desires, and intentions of the agent. In a pure goal-oriented semantics two instance variables of type *wff* can be defined, named *permanent-goals* and *actual-goals*, holding the formula for the permanent and actual goals. Usual UML fields can be defined for the specification of a plain object oriented agent, i.e. an agent implemented on top of e.g. a Java-based agent platform. However in different design stages different kinds of agents can be appropriate, on the conceptual level BDI agents can be specified implemented by a Java-based agent platform, i.e. refinement steps from BDI agents to Java agents are performed during the agent development.

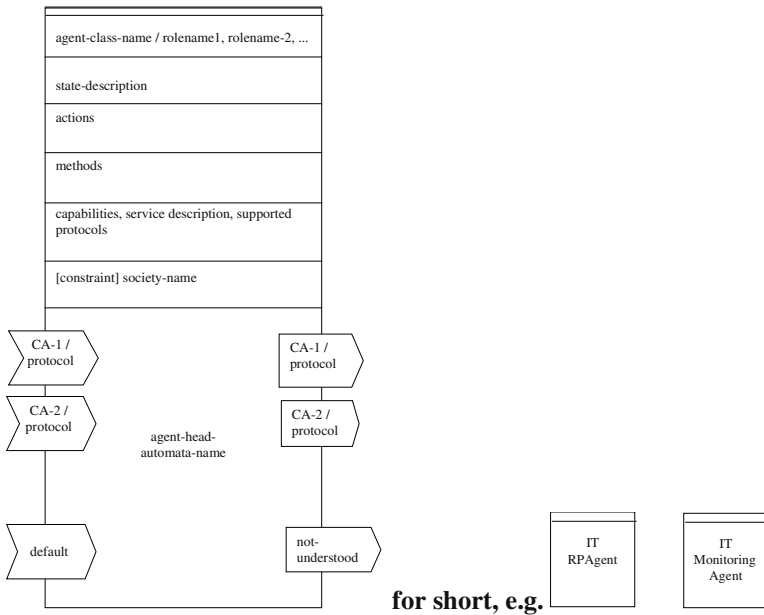


Fig. 18. Agent class diagram and its abbreviations

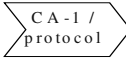
Actions: Pro-active behavior is defined in two ways, using *pro-active actions* and pro-active agent state charts. The latter one will be considered later. Thus two kinds of actions can be specified for an agent: pro-active actions (denoted by the stereotype <<pro-active>>) are triggered by the agent itself, if the pre-condition of the action evaluates to true. *re-active actions* (denoted by the stereotype <<re-active>>) are triggered by another agent, i.e. receiving a message from another agent. The description of an agent's actions consists of the action signature with visibility

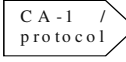
attribute, action-name and a list of parameters with associated types. Pre-conditions, post-conditions, effects, and invariants as in UML define the semantics of an action.

Methods: Methods are defined as in UML, eventually with pre-conditions, post-conditions, effects and invariants.

Capabilities: The capabilities of an agent can be defined either in an informal way or using class diagrams e.g. defining FIPA-service descriptions.

Sending and Receiving of Communicative Acts: Sending and receiving communicative acts characterize the main interface of an agent to its environment. By communicative act (CA) the type of the message as well as the other information, like sender, receiver or content in FIPA-ACL messages, is covered. It is assumed that classes and objects represent the information about communicative acts. The

incoming messages are drawn as  and the outgoing messages are drawn

as . The received or sent communicative act can either be a class or a concrete instance. The notation *CA-I / protocol* is used if the communicative act of class *CA-I* is received in the context of an interaction protocol *protocol*. In the case of an instance of a communicative act the notation *CA-I / protocol* is applied. As alternative notation *protocol[CA-I]* and *protocol[CA-I]* can be used. The context *protocol* can be omitted if the communicative act is interpreted independent of some protocol. In order to react to all kinds of received communicative acts, we use a distinguished communicative act *default* matching any incoming communicative act. The *not-understood* CA is sent if an incoming CA cannot be interpreted.

Matching of Communicative Acts: A received communicative act has to be matched against the incoming communicative acts of an agent to trigger the corresponding behavior of the agent. The matching of the communicative acts depends on the ordering of them, namely the ordering from top to bottom, to deal with the case that more than one communicative act of the agent matches an incoming message. The simplest case is the default case, *default* matches everything and *not-understood* is the answer to messages not understood by an agent. Since instances of communicative acts are matched, as well as classes of communicative acts, free variables can occur within an instantiated communicative act. This matching is formally defined in [14].

3.4 Ontologies

As we have already noticed e.g., in PASSI, several research approaches are dealing with the definition of ontologies using UML class diagrams [19][20], not only from the agent-oriented research community but also from the Semantic Web community [17] [22]. Bergenti et al. [19] take a pragmatic view an ontology definition applying UML class diagrams as shown in Fig. 19, defining the entities and on the other relating it to specific agents.

Cranefield et al. use UML to define agent communication languages (ACL) and content languages, like an object-oriented implementation of the FIPA ACL or FIPA SL [21], see also Fig. 20. They also apply UML for ontology definition and rely description logic with UML in [20].

In [22] an extension of UML is defined to cover DAML defining a high-level mapping between UML and DAML, e.g. Ontologies are viewed as packages, classes as classes, properties as attributes, associations and classes.

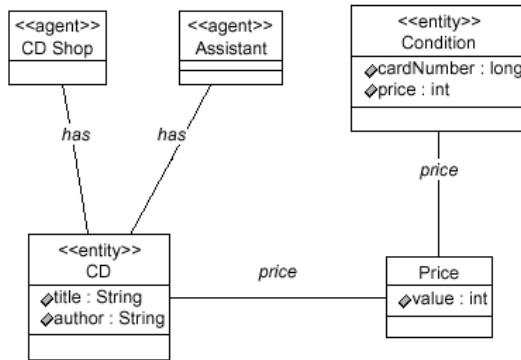


Fig. 19. UML-based ontology definition

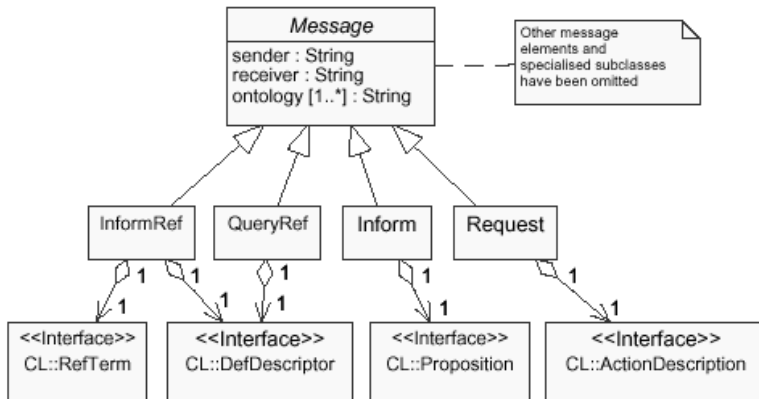


Fig. 20. Excerpt of object-oriented design of FIPA ACL/SL

3.5 Goals and Plans

Goals and plans are described by state charts or activity diagrams in several methodologies (see above). In [23] Huget uses UML 2.0 activity diagrams for the descriptions of goals and plans. We present here his example of a goal diagram corresponding to the interaction between the customer and the order acquisition (see Fig. 21).

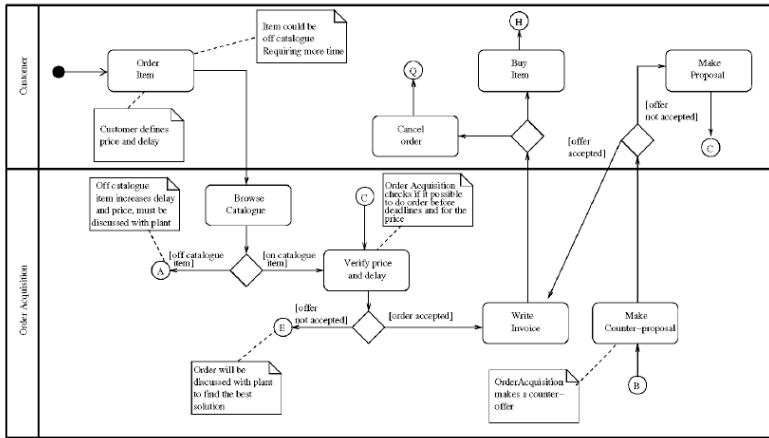


Fig. 21. Supply Chain Management scenario as Activity Diagram

The goal diagram expresses the following. Customer first performs the action *Order Item*. This ordered item is received by the Order acquisition. The Order acquisition checks if the ordered item is on catalogue (action *Browse catalogue*). If the ordered item is off catalogue, then the following of the actions is on A³. This characteristic only changes how the item is produced and priced. If the ordered item is in the catalogue, the order acquisition checks the price and the delay (action *Verify price and delay*). If the proposal made by the customer cannot be processed for this delay and price then the order acquisition goes to E. After several actions, the order acquisition comes back to B to make a counter-proposal which is accepted or not by the customer. If the customer accepts the counter offer, next action is to write an invoice (action *Write invoice*). If the customer does not accept, it can make another proposal. The following is as defined above. Finally, after writing the invoice, the customer has two choices: either accepting the order (action *Buy item*) or canceling the order (action *Cancel order*).

4 Conclusions and Further Research

In this paper we surveyed a number of important research contributions in the area of methodologies and notations for the development of agent-based systems based on UML. The general approach of building agent-based features on top of an established object-oriented model introduces a number of trade-offs, in particular regarding the natural design of agent-based systems. Yet, the large advantage of these approaches is that they fit easier into the object-oriented conception, and that it is relatively easy to present higher-quality tools by extending existing object-oriented tools. It appears that while objects and agents are certainly different notions (see e.g., the discussion in [41]), agent-oriented software engineering can greatly benefit from OO technologies and approaches. In particular, agent-oriented approaches are also suitable for areas

³ There exists only one matching for a letter: one encircled letter A with incoming arrow on this figure and one encircled letter A with outgoing arrow defined elsewhere

where object-oriented modeling has shortcomings. Here the abstractions inherent to agent-oriented software engineering can help us to overcome the limitations of the object-oriented approach.

Acknowledgement. We would like to thank the AOSE chairs for the invitation to contribute to this workshop proceedings. Moreover we want to thank all the people involved within FIPA TCs dealing with methodologies and notation for preparing an excellent collection of state-of-the-art paper on the AUML web-site.

References

- [1] Bauer, B., Müller, J.P., Odell, J.: Agent UML: A Formalism for Specifying Multiagent Software Systems, International Journal on Software Engineering and Knowledge Engineering (IJSEKE), Vol. 11, No. 3, pp.1–24, 2001 Engineering, 2001.
- [2] Kinny, D., Georgeff, M., Rao, A.: A Methodology and Modeling Technique for Systems of BDI Agents, in Proceedings of the Seventh European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW 96), LNAI 1038, Springer, 1996.
- [3] Kinny, D. and Georgeff, M.: A design methodology for BDI agent systems. Technical Report 55, Australian Artificial Intelligence Institute, Melbourne, Australia, 1995.
- [4] Kinny, D. and Georgeff, M.: Modelling techniques for BDI agent systems. Technical Report 54, Australian Artificial Intelligence Institute, Melbourne, Australia, 1995.
- [5] Kinny, D. and Georgeff, M.: Modelling and Design of Multi-Agent Systems, Proc. ATAL 96, 1996.
- [6] MESSAGE web site: <http://www.eurescom.de/public/projects/P900-series/p907/>
- [7] Giovanni Caire , Wim Coulier , Francisco Garijo, Jorge Gomez, Juan Pavon , Philippe Massonet, Francisco Leal, Paulo Chainho , Paul Kearney, Jamie Stark, Richard Evans , Agent Oriented Analysis using MESSAGE/UML, Proceedings AOSE 2001, Springer 2001.
- [8] Elisabeth A. Kendall, Margaret T. Malkoun, and Chong Jiang. A methodology for developing agent based systems for enterprise integration. In D. Luckose and Zhang C., editors, Proceedings of the First Australian Workshop on DAI, Lecture Notes on Artificial Intelligence. Springer-Verlag: Heidelberg, Germany, 1996.
- [9] Jürgen Lind: Iterative Software Engineering for Multiagent Systems: The MASSIVE Method. Springer, 2001
- [10] Ralf Kühnel, Agentenbasierte Software - Methode und Anwendungen, Addison-Wesley, 2000.
- [11] Bauer, B.; Bergenti, F., Massonet, Ph., Odell, J.: Agents and the UML: A Unified Notation for Agents and Multi-Agent Systems, Proceeding AOSE 2001, Montreal, Springer, 2001.
- [12] Bauer, B.; Müller, J. P.; Odell, J.: An Extension of UML by Protocols for Multiagent Interaction, Proceeding, Fourth International Conference on MultiAgent Systems, ICMAS 2000, Boston, IEEE Computer Society, 2000.
- [13] Marc-Philippe Huget (editor): FIPA-Modelling – Interaction Diagrams, first draft, online available at www.fipa.org
- [14] Bauer, B.: UML Class Diagrams Revisited in the Context of Agent-Based Systems, in Proceedings AOSE 2001, Montreal, Springer, 2001.
- [15] Wagner, G., *The Agent-Object-Relationship Metamodel: Towards a Unified View of State and Behavior.*, Technical Report, Eindhoven Univ. of Technology, Fac. of Technology Management, May 2002.

- [16] PASSI website: www.csai.unipa.it/passi
- [17] M. Cossentino, C. Potts: A CASE tool supported methodology for the design of multi-agent systems, in Proc. The 2002 International Conference on Software Engineering Research and Practice (SERP'02) Las Vegas (NV), USA, 2002.
- [18] W3C Note NOTE-rdf-uml-19980804, available online at <http://www.w3.org/TR/1998/NOTE-rdf-uml-19980804/>
- [19] Federico Bergenti, Agostino Poggi: A Development Toolkit to Realize Autonomous and Inter-operable Agents, in Proc. Autonomous Agents 2001, 2001.
- [20] Stephen Cranefield, Stefan Haustein, Martin Purvis: UML-Based Ontology Modelling for Software Agents.
- [21] Stephen Cranefield, Martin Purvis: Generating ontology-specific content languages
- [22] Baclawski, K., Kokar, M., Kogut, P., Hart, L., Smith, J., Holmes, W., Letkowski, J., and Aronson M., "Extending UML to Support Ontology Engineering for the Semantic Web." *Proc. of the Fourth International Conference on UML (UML2001)*, Toronto, October 2001
- [23] Marc-Philippe Hugot: Representing Goals in Multi-Agent Systems, unpublished paper, 2003
- [24] Mark F. Wood Scott A. DeLoach An Overview of the Multiagent Systems Engineering Methodology, In: Proceedings of the First International Workshop on Agent-Oriented Software Engineering, P. Ciancarini, M. Wooldridge, (Eds.) LNCS. Vol. 1957, Springer, 2001.
- [25] Wood, M. F.: Multiagent Systems Engineering: A Methodology for Analysis and Design of Multiagent Systems. MS thesis, AFIT/GCS/ENG/00M-26. School of Engineering,
- [26] DeLoach, S. A., Wood M. F.: Multiagent Systems Engineering: the Analysis Phase. Technical Report, Air Force Institute of Technology, AFIT/EN-TR-00-02, June 2000.
- [27] Paolo Bresciani, Paolo Giorgini, Fausto Giunchiglia, John Mylopoulos , and Anna Perini. TROPOS: An Agent-Oriented Software Development Methodology. Journal of Autonomous Agents and Multi-Agent Systems. 2003. Kluwer Academic Publishers (to appear).
- [28] Ivar Jacobson, Grady Booch, James Rumbaugh: The Unified Software Development Process, Addison Wesley, 1998.
- [29] J. Mylopoulos, M. Kolp, and J. Castro. UML for agent-oriented software development: The Tropos proposal. In Proc. of the 4th Int. Conf. on the Unified Modeling Language UML'01, Toronto, Canada, Oct. 2001
- [30] Tropos web site <http://www.cs.toronto.edu/km/tropos/>
- [31] GRL web site: <http://www.cs.toronto.edu/km/GRL/>
- [32] i* web site: <http://www.cs.toronto.edu/km/istar/>
- [33] J. Castro, M. Kolp and J. Mylopoulos. Towards Requirements-Driven Information Systems Engineering: The Tropos Project. Information Systems, Elsevier, Amsterdam, The Netherlands, 2002.
- [34] P. Busetta, R. Rönquist, A. Hodgson, and A. Lucas. JACK Intelligent Agents – Components for Intelligent Agents in Java. Technical Report TR9901, AOS, January 1999. <http://www.jackagents.com/pdf/tr9901.pdf>.
- [35] Lin Padgham and Michael Winikoff: Prometheus: A Methodology for Developing Intelligent Agents, In: Proceedings of AOSE 2002, Springer, 2002.
- [36] Prometheus home page: <http://www.cs.rmit.edu.au/agents/SAC/methodology.shtml>
- [37] Lin Padgham and Michael Winikoff, Prometheus: A Pragmatic Methodology for Engineering Intelligent Agents. In Proceedings of the workshop on Agent-oriented Methodologies at OOPSLA 2002. November 4, 2002
- [38] H. Van Dyke Parunak and James Odell. Representing Social Structures in UML, In: Proceedings of AOSE 2001, Springer, 2001.
- [39] Kennedy Carter eXecutable UML: <http://www.kc.com/MDA/xuml.html>
- [40] Model-driven Architecture: <http://www.omg.org/mda/>

- [41] Wooldridge, M.J. An introduction to multiagent systems. John Wiley & Sons, 2002.
- [42] Bauer, B. and Müller, J.P.: Agent-Oriented Software Engineering: Methodologies and Modeling Languages - A State of the Art Survey -, to be published as book chapter, 2003.
- [43] Wagner, G., A UML Profile for External AOR Models in Proceedings AOSE 2002, Springer, 2002
- [44] Luck, M., McBurney P., and Preist, C., eds. Agent Technology: Enabling Next Generation Computing. AgentLink, <http://www.agentlink.org>, 2003.

Towards a Recursive Agent Oriented Methodology for Large-Scale MAS

Adriana Giret and Vicente Botti

Departamento de Sistemas Informáticos y Computación,
Universidad Politécnica de Valencia, Spain
46020 Valencia, Spain
{agiret,vbotti}@dsic.upv.es

Abstract. Current business trends, policy markets, production requirements, etc., have created the need for integrating Multi Agent Systems (MAS). In the agent-specialized literature, we have found very little work about MAS methodologies that allow us to build MAS which is made up of two or more MASs. We think that several difficult challenges for automated systems can be tackled by giving full meaning to the MAS concept: adopting a recursive definition of MASs. In this work we outline the basis for a recursive agent oriented methodology for large-scale MAS.

1 Introduction

Nowadays arises the need to integrate pre-existent Multi Agent Systems (MAS) in domains where these integration and/or cooperation are imposed by business trends, policy markets, production requirements, etc. The question arises as to whether a MAS can be a collection of several interacting MASs, a hierarchy, or some other type of organization.

A large-scale MAS encompasses multiple types of agents and may, as well, encompass multiple MASs, each of them having distinct agency properties. A large-scale MAS needs to satisfy multiple requirements such as reliability, security, adaptability, interoperability, scalability, maintainability, and reusability. But, how can we specify and model the agency properties of a MAS made up of others MASs? We believe that MAS have to integrate recursive aspects to be able to comply with these requirements.

MAS offers powerful tools to realize complex problem spatialized solving or simulations systems. Some of these problems present hierarchical and multi-scale requirements and evolve in structured environments which posses recursive properties. In the intelligent manufacturing field, the need for some kind of hierarchical aggregation in real world systems has been recognized. These systems have to remain readable while they are expanded in a wide range of temporal and spatial scales. For example, a modern automobile factory, incorporates hundreds of thousands of individual mechanisms (each of which can be an agent) in hundreds of machines which are grouped in to dozens or more production lines. Engineers can design, build, and operate such complex systems by shifting

from the mechanism, to the machine or to the production line (depending on the problem at hand) and by recognizing the agents of higher levels as aggregations of lower-level agents. Also, in e-commerce applications, an enterprise is a legal entity which is independent of the individual people whose are its employees and directors.

In the agent-specialized literature, we have found very little work about methodologies which allow us to carry out recursive and dynamic analysis, design, and implementation of MAS. Most of the current approaches start from an atomic agent definition such as an indivisible entity and build MAS as compositions of interacting agents. Nevertheless, there are some works in which a nested MAS structure is included. Parunak and Odell, in [1], proposed UML conventions and AUML extensions to support nested agents' groups. Wagner, in [2], models an institutional agent which is made up of agents themselves.

In this work we try to define a set of concepts to help in the construction of large-scale MASs. The aim of this paper is to introduce a recursive agent model and an outline of a recursive MAS methodology. We are convinced that several difficult challenges for automated systems may be tackled by giving full meaning to the agent concept: adopting an abstract recursive agent definition. To this end we present the definition for abstract recursive agents in section 2. In section 3, we show an example of a large-scale MAS modelled as a recursive MAS. In section 4 we introduce an outline of a recursive agent oriented methodology. Finally we state our conclusions in section 5.

2 Abstract Recursive Agent

With a recursive approach to develop Multi Agent Systems as systems in which their components may be MASs themselves, the idea is as follows [1]. When we begin to analyze a group of agents (MAS) A , we identify the agents $\{a_1, a_2, \dots, a_n\}$ which execute certain functions. These agents may encapsulate individual persons, physical, or software entities (agents). They may also be other groups of MAS, say B , so we can have $a_i = B_i$, which we treat as black boxes. We can take this perspective as long as our analysis can ignore the internal structure of the member groups (MAS). However, subsequent analysis generally needs to 'open' these black boxes and look inside them to see the agent components and their corresponding functions; for example, when analyzing B we have that $B = \{b_1, b_2, \dots, b_m\}$. At this point, we insist on identifying which of B 's member agents is actually responsible for filling B 's role in A .

To support these ideas it seems appropriate to provide an abstract recursive agent (A-Agent) definition which will allow us to build Multi Agent Systems. This definition is based on the widely known agent definition of Wooldridge and Jennings [3].

Definition 1. *An A-Agent is a software system with a unique entity, which is located in some environment, which as a whole, perceives its environment (environment sensitive inputs). From these perceptions, it determines and executes*

actions in an autonomous and flexible way - reactive and proactive. These actions allow the A-Agent to reach its goals and to change its environment. From a structural point of view, an A-Agent can be an agent (atomic entity); or it can be a Multi Agent System (with a unique entity) made up of A-Agents which are not necessarily homogeneous.

An A-Agent is in a higher conceptual abstraction level than an agent. An A-Agent can be seen as a MAS, an organization, a federation or an institution with the added value that it can also be a composition of all this abstraction models. Furthermore, when we define two interacting A-Agent we could also be modelling two interacting organizations, federations, MAS or institutions. An A-Agent will exist only at modelling stages, in the end (at coding stages) it will be replaced possibly by a group of agents or also by a single agent.

Definition 2. A Multi Agent System is made up of two or more A-Agents which interact to solve problems that are beyond the individual capabilities and individual knowledge of each A-Agent.

Definition 2 extends the traditional notion of MAS when indicating that a MAS is made up of MASs. This will allow us to build systems in which the building blocks are interacting MASs that work together to reach one or several global goals (the global goal is the goal of the system as a whole).

From definition 1 we have:

- A-Agent of level 0 is an agent.
- A-Agent of level 1 is a traditional MAS.
- A MAS is an A-Agent.
- A-Agent of level n is a MAS made up of interacting MASs.

The recursive structure of an A-Agent can be represented in a graphical way using UML as shown in Figure 1. This representation is similar to the one presented in [1] for the holonic perspective for agent oriented software engineering. In fact, our A-Agent structural definition is inspired by holonic concepts [4,5].

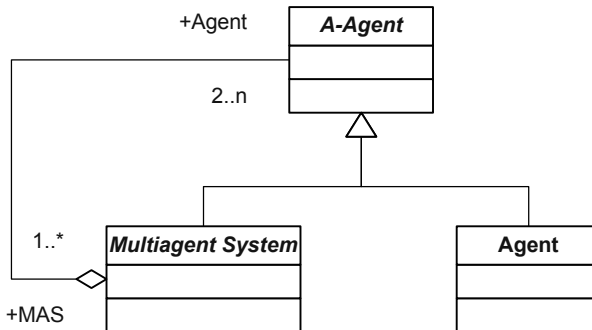


Fig. 1. Abstract Recursive Agent

Figure 1 shows that a MAS is made up of either: two or more agents or two or more MASs.

In [6], we have also proposed a formalization of MAS behaviours in terms of its constituent agent behaviour or MAS behaviour. In summary, the reactive behaviour of a MAS is determined by its perception which is defined as the union of the set of perceptions of its members, and by its actions, which in turn are defined as the union of the group actions executed by its members and the union set of the primitive actions carried out by each of its constituent entities. The intentional behaviour of a MAS is determined by its goals. These goals are defined as the union of the set of systems' goal derived from congruent patterns of interactions among its members and the set of joint goals of its constituent entities.

3 A Large-Scale MAS as a Recursive MAS

One of the most difficult challenges for automated systems is scalability and adaptation. In life systems there are many useful concepts, including examples on how to scale up, evolve, adapt, interoperate, organize, and so on. Complex and adaptive life systems are large, intricate and require active autonomous entities. Life systems are recursive and they enable the construction of very complex systems from more simple entities [5]. What about MASs? Are MAS MASs arranged in clusters, a hierarchy, or some other type of organization? In this section, we present an example in which the usefulness of our definition can be observed to describing complex large-scale problems with multiple levels of abstraction.

Let us suppose a Multinational company, called AG, which has different National companies distributed among different countries. The objective is to model the multinational as a MAS.

Each National company can be an A-Agent since it has got agenthood characteristics. The National company is autonomous in its national environment; it acts in the national market with its own market and production rules. At the same time, it must be able to interact with other National companies to exchange materials, personnel, knowledge, etc. The National company, is also governed by the rules and norms of the Multinational for its international relations (other National companies).

The international companies' relationships define the rules, norms and policies of the multinational. In Figure 2(a), geographical areas can be observed in which the relationships among the national companies are narrower. In addition the commercial agreements among the different countries define new interrelation rules among the national companies of these zones. For example, in Europe, the European Union countries are governed by certain standards and norms of the community; and in South America the Southern Cone Common Market - MERCOSUR (Paraguay, Argentina, Chile, Brazil, Uruguay and Bolivia) and the countries of the Andean Community (Bolivia, Colombia, Ecuador, Peru and Venezuela). The relationships of the countries of these markets with other coun-

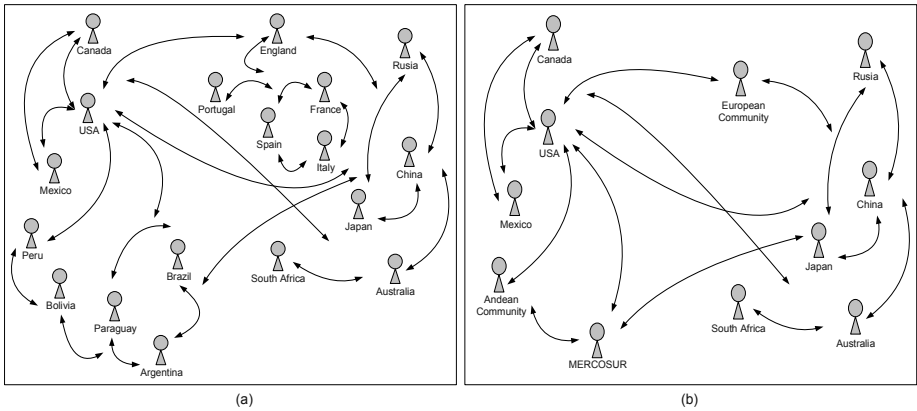


Fig. 2. Regional and National Companies of a Multinational Company.

tries or regional markets are managed by their local market rules. Each market can be modeled as an A-Agent. This generalization is shown in Figure 2(b). It is important to note that Bolivia, as a National Company, belongs to two Regional Companies (MERCOSUR and the Andean Community).

Up to this point, we have identified 4 levels of abstractions (Figure 3(a)): the Multinational company, the Regional companies, and the National companies. We have been able to model the Multinational as a MAS, which is composed by A-Agents that are related to each other with certain behavior patterns that define the Multinational company. If the National companies will be made up of agents (A-Agents of level 0), we can think of a National company as a traditional MAS (A-Agent of level 1), the Regional companies as A-Agents of level 2 and the Multinational as A-Agent of level 3.

Apart from modeling the outside relationships, if the designer's interest is also to model the internal structure of each National company, the National company should be observed from inside. Inside each National company there would be new distributed companies in different cities or with autonomy for certain activities. In turn, each Local company is subordinated to the National company and each National one to the Multinational. Thus we have a new level of abstraction, the Local company as an A-Agent of level 1, the National company as an A-Agent of level 2, the Regional company as an A-Agent of level 3 and the Multinational as an A-Agent of level 4 Figure 3(b).

If the National company is not subdivided into city companies or autonomous companies. Then the National company is a traditional MAS composed of national domain-specific agents (A-Agents of level 0), which are interrelated agents and carry out specific functions. These national domain-specific agents define the services provided by the National company inside the country and outside the country. This very same analysis should be made for each Local company until we reach the agents, which define and implement the company as a whole. In summary, the final result of the analysis should be similar to Figure 3(c). In

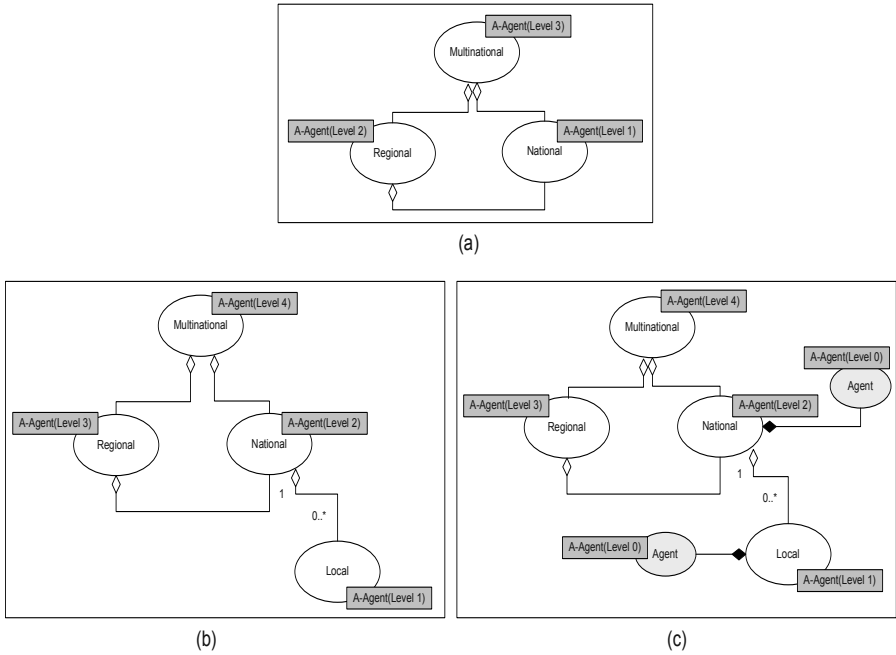


Fig. 3. Different levels of abstraction of the Multinational Company.

Figure 3(c), it can be observed that the National company is composed of zero or more Local companies, and each Local company, in turn, is an A-Agent of level 1.

Again, the Multinational can be considered from outside as an A-Agent, since it is located in an environment, the world market; it is autonomous; it has its own economic and market policies; it is social, i.e. it interacts with other entities for purchasing, selling, recruiting, leasing, etc.; it is pro-active, since for example according to the world market trends it is able to modify its current market policies, etc.

4 Recursive Agent Oriented Methodology

In this section we will introduce an outline of a recursive agent oriented methodology. This methodology will help us in the construction of MAS made up of pre-existing MASs or MAS made up of interacting MASs emerged from groups of agents which have close interactions and which implement a well defined functionality. As a result this group of close agents can be encapsulated as a new MAS (A-Agent) and hence modeled and implemented as an entity with its own characteristics.

The recursive agent oriented methodology tries to reduce the complexity of large-scale MAS, dividing the domain problem in simpler sub-problems and

considering every sub-problem as an A-Agent. Every such A-Agent can in turn be decomposed in simpler interacting A-Agents. Until we reach an abstraction level in which there is no more subdivision, that is all the constituent members of the MAS (A-Agent) are agents. We can think of the methodology as a MAS-driven methodology.

The recursive agent oriented methodology models each MAS dividing it in more concrete aspects that form different *views* of the system. This idea already appears in the work of Kendall [7], MAS-CommonKADS [8], and later in GAIA [9]. The way in which the views are defined is inspired by INGENIAS methodology [10].

To describe a MAS, the developer will use the following models:

- **A-Agent Model:** Describes agents and A-Agents, their task, goals, initial mental state, and played roles. Moreover, these models are used to describe intermediate states of agents and A-Agents. These states are presented using goals, facts, tasks, or any other system entity that helps in its state description. The constructs added to the INGENIAS notations for A-Agent and its associated abstract-tasks, abstract-goals and abstract-mental state are depicted in Figure 4. In contrast to INGENIAS in our approach a group (as well as an organization) of agents -an A-Agent- executes abstract-task, has abstract-goals and abstract-mental states. An A-Agent may play a role, but it can not execute task, can not have goals and mental states. All these abstracts entities have to be implemented by real entities. That is, for each A-Agent identified at every development step, there will be a group of agents that will implement the corresponding functionality, will execute the corresponding task, will have the corresponding goals and mental states. An abstract task will be implemented by a work flow. An abstract goal will possibly be replaced by a conjunction of goals or by a common goal and so on.

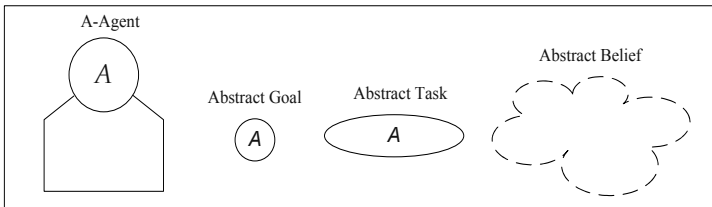


Fig. 4. Abstract Notations for the A-Agent Model.

In this work we focus on the A-Agent meta model. To define the A-Agent meta-model we add the gray entities and the bold lines identified in Figure 5 to the agent meta-model of INGENIAS. Following INGENIAS we use GOPRR (Graph, Object, Property, Relationship, and Role)[11] primitives in UML notation.

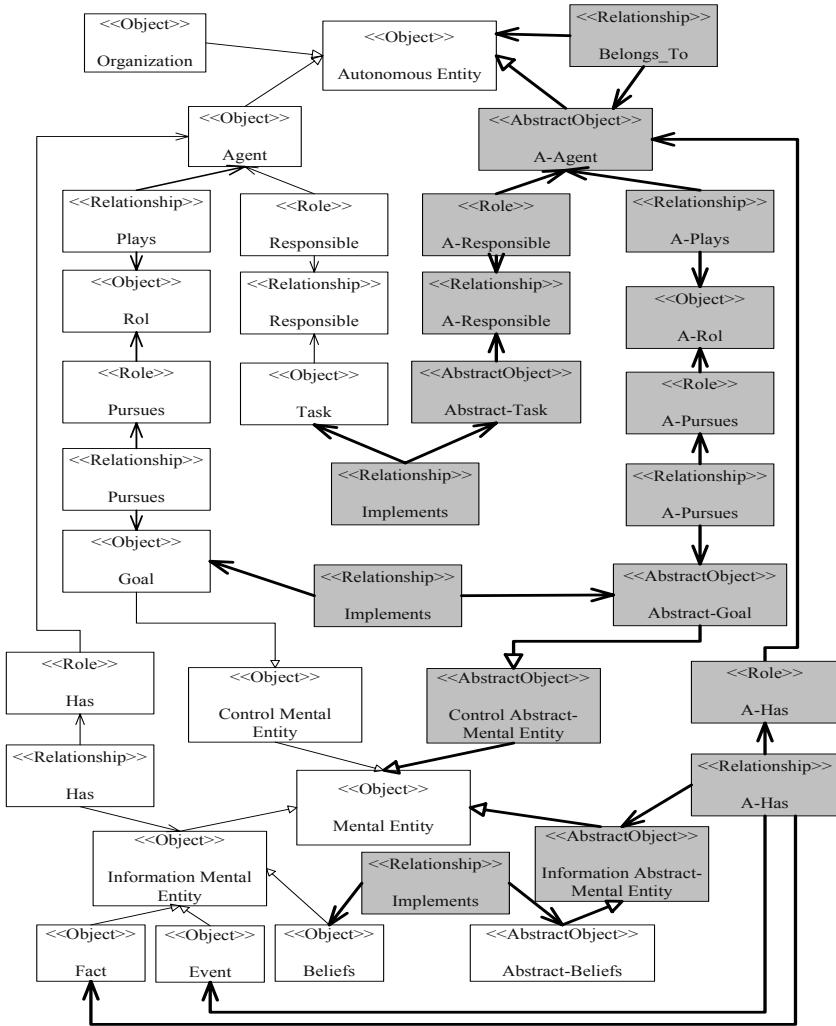


Fig. 5. Main Entities in the A-Agent meta-model.

- **Interaction model:** Describes how interaction among agents and A-Agents takes place. Each interaction declaration includes involved actors, goals pursued by interaction, and a description of the protocol that follows the interaction. Figure 6 depicts the abstract interaction notations. An abstract interaction is an interaction in which at least one of the speakers is an A-Agent.
- **Task-goals model:** Describes relationships among goals and tasks, goal structures, and task structures. It is also used to express which are the inputs and outputs of the tasks and what are their effects on environment

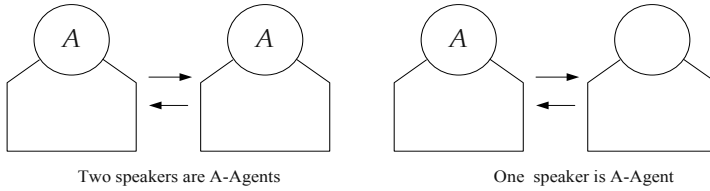


Fig. 6. Abstract Interaction Notation.

or on A-Agent's mental state. This model describes also the subdivision of the abstract task, identified in the A-Agent model, into a set of tasks.

- **Organization model:** Describes how system components (agents, A-Agents, roles, resources, and applications) are grouped together, which are executed in common, which goals they share, and what constraints exists in the interaction among agents and among A-Agents.
- **Environment model:** Defines agent's and MAS's perception in terms of existing elements of the system.

These models will help in the modelling of MASs of any level of abstraction (that is a conventional MAS -level 1- made up of agents, a MAS of level 2 made up of MAS of level 1, and so on). In every model the definition of the MAS behaviour is carried out following the work presented in [6].

The software development process guided by this methodology will be a recursive, incremental and concurrent MAS driven process, see Figure 7. It will have as many iterations as levels of abstractions are identified. The result of each iteration will be a MAS of level n in which its structure and functionality are defined in terms of the previous models. In each new iteration there will be as many concurrent processes as non defined MASs of level $n - 1$ of the previous iteration (because, we can have pre-existing MAS of level $n - 1$). Each iteration will be a recursive, incremental and concurrent process.

Each iteration will be conducted in the following way. In the analysis phase, organization models will be produced to sketch how the MAS looks like. In this step, we can identify potential constituent A-Agents (that is, group of agents with close interaction, with an identified functionality or goal, in which every member interacts to solve some problem, and with self-regulating rules of actions) or pre-existing constituent MASs. The models obtained in this phase will be refined later to identify common goals and relevant tasks to be performed by each A-Agent (task-goal model and environmental model). More details will be added specifying A-Agent interactions with interactions models and environmental models, and, as a consequence, refining A-Agents's mental state with A-Agent models. For each emerged MAS a new process is started, until we reach a step in which there are no more non specified MASs (A-Agents of level > 0).

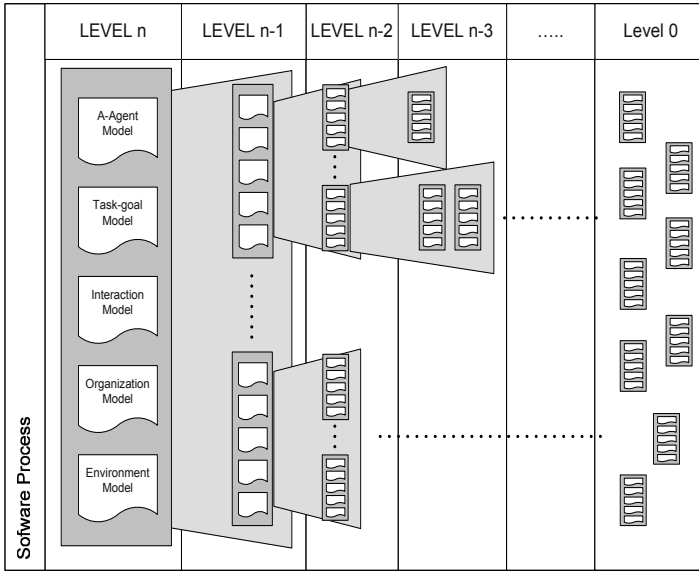


Fig. 7. Software Development Process.

5 Conclusion

In this work, we have presented a recursive approach to develop large-scale MASs as systems in which their components may be MASs themselves. We have presented in section 3 an example of a complex system modeled as a recursive MAS. In this example we have shown the advantages of modeling a large-scale MAS as a recursive MAS. Abstracting away from the complexity of the problem and focusing at each level in the interactions and the behaviour of each constituent entity.

In section 4, we have illustrated an outline of a recursive, incremental and concurrent agent oriented methodology. The recursive methodology models each MAS dividing it in more concrete aspects that form different *views* of the system. The way in which the views are defined is based in INGENIAS methodology [10]. We have presented a sketch of the A-Agent meta-model. The software process will have as many iterations as levels of abstractions are identified. The result of each iteration will be a MAS of level n in which its structure and functionality are defined in terms of: A-Agent models, Task-goal models, Interaction models, Organization models and Environment models. In each new iteration there will be as many concurrent processes as non defined MASs of level $n - 1$ of the previous iteration (because, we can have pre-existing MAS of level $n - 1$). Each iteration will be a recursive, incremental and concurrent process.

This paper is a preliminary report of our research. There are a lot of open problems. In [6], we pointed out a definition of MAS' behaviour in terms of its constituent members in a top-down fashion. We also need to formalize it in a

bottom-up way to deal with the definition of the behaviour of MAS' emerged from pre-existing agents. We have to complete the definition of the four models and meta-models to comply with the recursive properties in A-Agent structures. We have to formalize the data flows relations among the four models. There is no CASE tool to help in the software recursive process.

References

1. Parunak, V.D., Odell, J.: Representing social structures in UML. In Agent-Oriented Software Engineering II, M. Wooldridge, G. Weiss, and P. Ciancarini, eds. Springer Verlag (2002) 1–16
2. Wagner, G.: Agent-oriented analysis and design of organizational information systems. in J. Barzdins and A. Caplinskas (Eds.), Databases and Information Systems. Kluwer Academic Publishers. (2001) 111–124
3. Wooldridge, M., Jennings, N.R.: Intelligent agents - theories, architectures, and languages. Lecture Notes in Artificial Intelligence, Springer-Verlag. ISBN 3-540-58855-8 **890** (1995)
4. HMS, P.R.: HMS Requirements. HMS Server, <http://hms.ifw.uni-hannover.de/> (1994)
5. Koestler, A.: The Ghost in the Machine. Arkana Books (1971)
6. Giret, A., Botti, V.: Recursive agent. In Proceedings of Iberagents 2002, Agent Technology and Software Engineering (2002)
7. Kendall, E.: Developing agent based systems for enterprise integration. IFIP Working Conference of TC5 Special Interest Group on Architectures for Enterprise Integration (1995)
8. Iglesias, C., Garijo, M., Gonzalez, J., Velasco, J.: Analysis and design of multi agent systems using MAS-CommonKADS. In Singh, M. P., Rao, A., and Wooldridge, M.J. (eds.) Intelligent Agentd IV LNAI, Springer Verlag **1365** (1998)
9. Wooldridge, M., Jennings, N.R., Kinny, D.: The Gaia methodology for agent-oriented analysis and design. Journal of Autonomous Agents and Multi-Agent Systems **15** (2000)
10. Gomez, J., Fuentes, R.: Agent oriented software engineering with INGENIAS. In Proceedings of Iberagents 2002, Agent Technology and Software Engineering (2002)
11. Lyytinen, K.S., Rossi, M.: METAEDIT+ - a fully configurable multi user and multi tool CASE and CAME environment. Springer Verlag LGNS 1080 (1999)

Agent-Oriented Modeling by Interleaving Formal and Informal Specification

A. Perini¹, M. Pistore^{2,1}, M. Roveri¹, and A. Susi¹

¹ ITC-irst, Via Sommarive, 18, I-38050 Trento-Povo, Italy
{perini,roveri,susi}@irst.itc.it

² Department of Information and Communication Technology
University of Trento, via Sommarive 14, I-38050 Trento-Povo, Italy
pistore@dit.unitn.it

Abstract. The goal of this paper is to discuss possibilities of intermixing formal and informal specification in order to guide and support the conceptual modeling process in software development. We sketch a framework based on an agent-oriented methodology that provides a modeling language which allows for the definition of both informal and formal specification. We show how formal techniques can be used to guide and support the analyst while building and refining a conceptual model. Examples of its applications are discussed, with reference to the decision making process undertaken by the analyst when performing a set of activities relevant for requirements engineering, such as requirements elicitation and refinement, user validation of requirements specification, or management of requirements evolution. A case study taken from a technology transfer project in the agricultural domain is used to illustrate the approach.

1 Introduction

In the last years, a considerable effort in defining agent-oriented approaches to software development is going on [5,12,23]. The main reason for this can be drawn back to the recognition that the agent paradigm, beside providing a useful technology to build software systems with an open architecture, offers appropriate abstractions for specifying and designing critical properties of these system, such as the dynamic evolution of their architecture and the interaction protocols of system components.

Most of the proposed methodologies adopt visual modeling as a core process, according to a best practice in structured Object Oriented software development processes [15], as well as to the ideas proposed by the Agile software development approaches [1].

The usage of visual modeling languages in the software development process offers advantages such as that of providing an effective communication framework for different stakeholders of the process. For instance, a use-case model can be used to discuss system requirements with the users, or a structural pattern

allows the analyst to detect and discuss with the user inconsistencies emerging from the model, as proposed in [8]. Nevertheless, visual modeling languages which lack a formal definition of their semantic, can lead to subjective models, which can hardly be refined in a straightforward way into a system design. Moreover, a typical question when building a conceptual model is: “when can I stop refining it?” This weakness is usually addressed by structured methodologies, which provides guidelines for the analyst and for the designer in building, refining and documenting the process’ artifacts that are based on conceptual models, e.g., [15].

Formal specification languages solve some of the weaknesses of visual modeling languages, specifically, they permit to define models with a precise semantics, and facilitate their transformation into system designs. However, writing formal specifications usually requires strong skills, and formal specifications are often very ineffective for discussing with the stakeholders. Moreover, the formalization “a posteriori” of the visual languages provided by the conceptual modeling frameworks is not at all an easy task, due to the ambiguities in the interpretation of the graphical notations.

The basic idea discussed in this paper concerns the possibility to exploit formal techniques to guide and support the analyst while building and refining a conceptual model. In particular we will focus on the decision making process undertaken by the analyst while performing a set of activities, relevant for requirements engineering, such as requirements elicitation and refinement, or user validation of requirements specification. We will also analyze how results of deductive reasoning procedures run on a formal specification can support the analyst’s decisions. Our claim is that, if the diagrammatic models are equipped with a formal semantics, then only a limited formalization effort is necessary to exploit formal techniques. Moreover, the diagrammatic notations make it possible to interpret the models in an informal way, for instance when discussing with the stakeholders.

In our approach we refer to the *Tropos* methodology [3,19], an agent oriented software development methodology which provides a conceptual modeling language that can be used both to build an informal specification or a formal one [9, 11]. It is this common conceptual model that allows for the formal interpretation of the diagrammatic models that we have described above. *Tropos* proved to be effective especially in domain modeling and in system requirements analysis, early stages activities in the software development process where we intend to focus on as a starting point. Our long term objective is that of providing a tool that supports the analyst and the designer that use informal modeling, in performing the deductive reasoning on a formal specification which has been automatically derived from the informal model.

The paper is structured as follows. Section 2 recalls the basic features of conceptual modeling in *Tropos* and how to build informal and formal specification. Section 3, presents our approach to interleaving informal and formal specification with reference to specific requirements engineering activities. Related works

are discussed in Section 4. Finally, conclusions and future work are presented in Section 5.

2 Background

The *Tropos* methodology [3,19] is an agent-oriented software development methodology which provides a visual modeling language that can be used to define both an informal specification and a formal one. From a practical point of view, the methodology guides the software engineer in building an informal, conceptual model that is incrementally refined and extended from an early requirements model, namely a representation of the organizational setting where the system-to-be will be introduced, to system design artifacts, according to a requirements-driven approach.

The *Tropos* language allows to model intentional and social concepts, such as those of actor and goal, and set of relationships, such as actor dependency, goal decomposition, means-end and contribution relationships. These elements support the modeling of basic goal analysis techniques. The language ontology has been given in terms of common sense (informal) definitions. An *actor* models an entity that has strategic goals and intentionality, such as a physical agent, a role with respect to a given context, or a set of roles (i.e., a position). *Goals* represent the strategic interests of actors. Two basic type of goals are considered, namely hard and soft goals, the latter having no clear-cut definition and/or criteria as to whether they are satisfied. Softgoals are useful for modeling goal/plan qualities and non functional requirements. A *dependency* between two actors indicates that an actor depends on another in order to achieve a goal, execute a plan, or exploit a resource.

Basic modeling activities in *Tropos* include the identification of the actors with their goals and of the actors mutual dependencies. Each goal can be analyzed from the point of view of the individual actor considering: possible sub-goals (*AND decomposition*); means to satisfy these goals (*means-end relationship*); alternative ways to achieve a specific goal (*OR decomposition*); goals or plans or resources that can contribute positively or negatively to its achievement (*contribution*). All these models can be depicted using two basic types of diagrams, namely, actor and goal diagrams. A detailed account of modeling activities can be found in [3].

A *Tropos* specification provides a “static” view of the organizational setting and of the dependencies among the different elements of the domain. A Formal *Tropos* (*FT* hereafter on) specification [9,11] extends a *Tropos* specification with annotations that characterize the valid behaviors of the model. In *FT* the emphasis goes in modeling the “strategic” aspects of the evolutions of the model. Thus, an *FT* specification consists of a sequence of class declarations such as entities, actors, goals, and dependencies. These are the formal counterparts of the elements of the “informal” *Tropos* specification. Each declaration associates a set of attributes to the class and characterizes its instances. Moreover, class declarations contain temporal constraints expressed in a typed first-order linear

time temporal logic (LTL). These constraints describe the valid lifetime evolutions of the model in terms of temporal evolutions of set of instances of the classes in the specification. Two critical moments in the life-cycle of goals and dependencies are the instants of their *creation* and *fulfillment*. The creation of a goal is interpreted as the moment in which the owner or depender expects or desires to achieve the goal, while its fulfillment is the moment in which the goal condition is actually achieved. In *FT*, creation and fulfillment constraints can be used to define conditions for these two moments in the life of intentional elements. Creation and fulfillment conditions are used, e.g., for defining constraints on the lifetimes of sub-goals in a goal decomposition (sub-goals are created after the parent goal and should be fulfilled before the parent goal can be fulfilled), or for defining the responsiveness of an actor w.r.t. the dependencies (an actor can take care immediately of some of them while delaying other dependencies). *FT* also provides *invariant* constraints that define conditions that should be true throughout the lifetime of class instances. Typically, invariants define relations on the possible values of attributes, or cardinality constraints on the instances of a given class.

Given an *FT* specification, one can ask questions such as: Can we construct valid operational scenarios based on the model? Is it possible to fulfill the primary goals of actors in the current model? Do the decomposition links induce a meaningful temporal order for goal fulfillment? Do the dependencies represent a valid synergy or synchronization between actors? These questions can be formulated as formal queries on a *FT* model and answered by the T-TOOL [10], an automatic verification tool based on the NUSMV [6] model checker (see [9, 10] for more details).

The effectiveness of the *Tropos* (and of the *FT*) methodology has been illustrated by several case studies [10,11,19,20]. In the following we will introduce a simple example giving both the informal specification and the formal one. The example is extracted from a real case-study developed in a technology transfer project in the domain of decision-support systems in agriculture, described in [20]. In particular, we will focus on goal modeling, represented by a simple goal diagram, and we will describe basic questions that the analyst can issue during informal modeling and that can be automatically answered when adding formal annotations.

2.1 Informal Modeling

The example considered corresponds to a fragment of the early requirements model for the agriculture domain. The early requirements model in *Tropos* describes the domain stakeholders (modeled as actors), their goals, and the mutual dependencies. In our case, the actor *Producer* represents the apple grower who pursues objectives such as applying Integrated Production (IP)¹ techniques with

¹ Integrated Production (IP) aims at a sustainable approach to agriculture production. In plant disease control, it promotes the use of low-impact techniques and chemicals, and the exploitation of natural defense mechanisms.

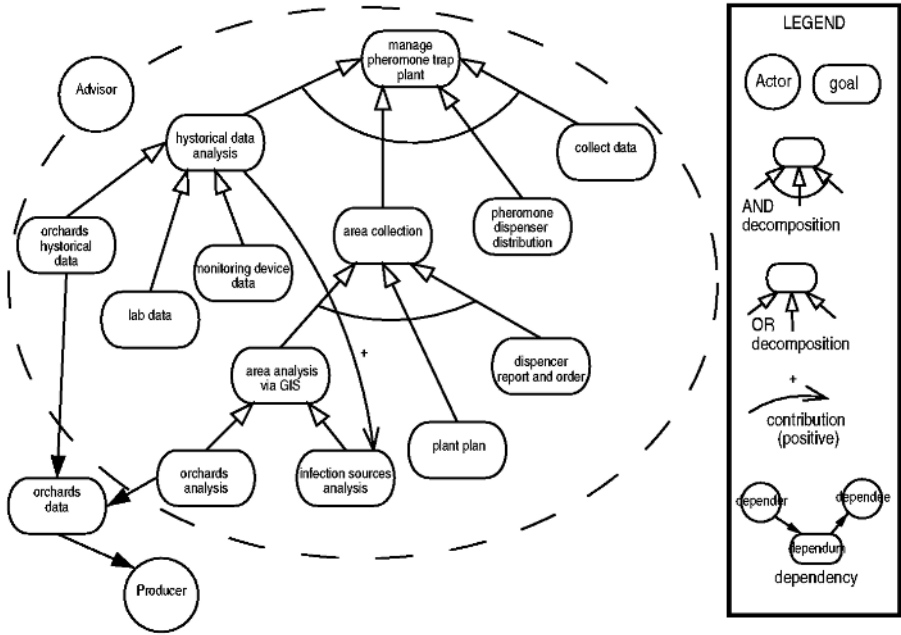


Fig. 1. Goal diagram of the actor Advisor, showing an example of goal analysis.

the help of agronomists of the local advisory service, represented by the actor Advisor. The example focuses on a particular technique for reducing the risk of infection of an apple pest which requires to install in the orchard a pheromones trapping system for preventing the pest population growth in the area. The design of the trapping system needs to take into account the geometry of the field (e.g., perimeter), geographical feature of the area of the field and the possible infections sources in the neighborhoods.

Figure 1, shows goal analysis of the Advisor, as resulting from a set of interviews to agronomists, concerning approaches currently in use for applying pheromones trapping techniques.

During goal analysis the analyst intends to extract all the possible steps that the Advisor has to fulfil to achieve the higher level goal manage pheromone trap plant. The analysis starts from the decomposition of this goal manage pheromone trap plant in a set of goals that has to be all satisfied (AND decomposition). An important step is the analysis of the agronomical history of the area of interest (modeled with the goal hystorical data analysis), that can be done by finding the data directly from the orchards descriptions (orchards hystorical data) maintained by the producers, or by accessing the data from the research laboratory (lab data), or by using data coming from the orchard monitoring systems, like meteorological data, (see the goal monitoring devices data). In this case the OR decomposition allows the analyst to model alternative ways to access data that can be available from different information sources. Crucial for the advisor is the possibility to

```

Actor Advisor
Actor Producer
Goal ManagePheromoneTrapPlant
  Actor Advisor
  Mode achieve
Goal AreaCollection
  Actor Advisor
  Mode achieve
  Attribute constant mptp : ManagePheromoneTrapPlant
  Creation condition  $\neg$  Fulfilled(mptp)
  Invariant mptp.actor = actor
  Fulfillment condition
     $\exists$  aavg : AreaAnalysisViaGIS ((aavg.actor = actor)  $\wedge$  Fulfilled(aavg))  $\wedge$ 
     $\exists$  pp : PlantPlan ((pp.actor = actor)  $\wedge$  Fulfilled(pp))  $\wedge$ 
     $\exists$  drao : DispenserReportAndOrder ((drao.actor = actor)  $\wedge$  Fulfilled(drao))
Goal OrchardAnalysis
  Actor Advisor
  Mode achieve
  Attribute constant aavg : AreaAnalysisViaGIS
  Creation condition  $\neg$  Fulfilled(aavg)
  Invariant aavg.actor = actor
  Fulfillment condition
     $\exists$  od : OrchardsData ((od.depender = actor)  $\wedge$  Fulfilled(od))
Goal Dependency OrchardsData
  Depender Advisor
  Dependee Producer
  Mode achieve
  Creation condition
     $\exists$  ohd : OrchardHistoricalData ((ohd.actor = depender)  $\wedge$   $\neg$  Fulfilled(ohd))  $\vee$ 
     $\exists$  oa : OrchardAnalysis ((oa.actor = depender)  $\wedge$   $\neg$  Fulfilled(oa))
  Invariant  $\exists$  ohd : OrchardHistoricalData (ohd.actor = depender)  $\vee$ 
     $\exists$  oa : OrchardAnalysis (oa.actor = depender)

```

Fig. 2. Excerpt of *FT* specification.

collect information about geographical and biological characteristics for the areas that seems to be candidate for the application of the pheromone trapping techniques (modeled with area collection). This goal can be accomplished by satisfying three sub-goals, namely, the geographic analysis of the areas (area analysis via GIS), the planning of the pheromone system (plant plan) and the distribution of the pheromone dispensers (dispenser report and order). All these goals have to be satisfied in order to obtain the set of needed information on the area. The dispenser distribution and the monitoring of the results obtained upon application of the techniques complete the accomplishment of the higher level goal.

2.2 Formal Modeling

An excerpt of the *FT* specification for the agriculture example is depicted in Figure 2. The *FT* specification can be obtained from the *Tropos* model by mapping actors and intentional elements into corresponding *FT* classes. The attributes in *FT* are references to other classes. For example, goal *OrchardAnalysis* refers to the *AreaAnalysisviaGIS* goal that motivates the advisor to get the orchard data (attribute *aavg*). Similarly, dependency *OrchardsData* refers to *OrchardHistoricalData* and to *OrchardAnalysis* goals of the advisor. Since actor *Advisor* is the

owner of goals `ManagePheromoneTrapPlant`, `AreaCollection` and `OrchardAnalysis`, the *FT* specification has `Advisor` as the **Actor** attribute of the three goals. Similarly, **Depender** and **Dependee** attributes of dependencies represent the two parties involved in a delegation relationship. Goals and dependencies in Figure 2 are also equipped with a mode attribute, which defines the modality of fulfillment. The mode of goal `ManagePheromoneTrapPlant`, for instance, is **achieve**, which means that the advisor wants to reach a state where he was able to manage the pheromone trap plant, and therefore the goal is fulfilled. Another modality in **maintain**, where the fulfillment condition is to be continuously maintained. Figure 2 contains also some examples of constraints on the lifetime of class instances. For instance, the first invariant of Figure 2 binds a `AreaCollection` object with its father `ManagePheromoneTrapPlant` object. Typically, primary intentional elements, (e.g., `ManagePheromoneTrapPlant`) have fulfillment constraints, but no creation constraints: we are not interested in modeling the reasons why an advisor wants to manage a pheromone trap plant. Subordinate intentional elements (e.g., `AreaCollection`, `OrchardAnalysis`) typically have constraints that relate their creation with the state of their parent intentional elements. For instance, Figure 2 shows that a creation condition for an instance of goal `AreaCollection` is that the parent goal `ManagePheromoneTrapPlant` is not yet fulfilled: if the advisor has already managed the pheromone trap plant there is no need to manage it again (unless a new management activity is started). The creation condition of dependency `OrchardsData` together with the fulfillment condition of goals `Orchard-HistoricalData` and `OrchardAnalysis` elaborate the delegation relationship between `Advisor` and `Producer` in the corresponding *Tropos* diagram.

In an *FT* specification we can also specify properties desired to hold in the domain, so they can be verified against the model we built. In *FT* we distinguish between **assertion** properties that should hold for all valid evolutions of the *FT* specification, and **possibility** properties that should hold for at least one. Given the *FT* specification and a set of properties, we can verify whether the *FT* specification satisfies the properties by means of the T-TOOL.

3 The Framework

The proposed framework is based on the idea of exploiting formal techniques to guide and support the analyst while building and refining a conceptual model. We describe it focusing on a set of requirements engineering activities, such as requirements elicitation and refinement, or user validation of requirements specification. These activities involve domain stakeholders and an informal analyst (called from now on *iAnalyst*), which can pose specific questions to a formal analyst (*fAnalyst*)². After a preliminary acquisition of information on domain stakeholders, on their goals and on their reciprocal dependencies, the *iAnalyst*

² Our long term goal is that of deriving the requirements of a CASE tool that could play the role of the *fAnalyst*, as emerging from the following discussion. For the moment, the role of the *fAnalyst* has to be played by a human actor.

starts building an early requirements model, such as that described by the diagram depicted in Figure 1.

Once the preliminary early requirements model has been devised, the iAnalyst proceeds with the analysis of this model. Covered activities are: the *assessment of the model* against possible inadequacies, incompleteness or inconsistencies; the *validation of the resulting specification* with the stakeholders in order to end up with an agreed set of requirements; and, when inconsistencies are discovered the needs of model revision, the *management of model refinement and evolution*, in order to maintain its consistency, propagate changes, merge redundant information.

In these activities, the iAnalyst has the possibility to make queries on specific aspects of the model that the fAnalyst translates into *FT* properties and checks on the *FT* model. The queries of the iAnalyst may concern different aspects of the model. In particular, the iAnalyst can check:

- the capability of an actor to fulfill a given goal, possibly assuming that some sub-goals or some dependencies cannot be achieved;
- the presence of constraints on the temporal order in which activities/goals can be started and/or achieved;
- the presence of implicit cardinality constraints on the relationships among goals in a goal diagram;
- the fact that a given property is respected by all the valid behaviors of the model;
- the fact that a given property is exhibited by some of the valid behaviors of the model.

In most of the cases, the formalization effort required to the fAnalyst for answering to these queries is small. The *FT* model can be obtained by an automatic translation of the informal model [10], and the *FT* property is obtained directly by the informal query of the iAnalyst.

In the following, we discuss three examples that illustrate the framework with respect to the different requirements engineering activities. In each example we identify a set of specific questions the iAnalyst can be interested in; we formalize these questions using a high level query language; we show how to map these queries into low-level *FT* properties that are checked by the T-TOOL; and we discuss the answers obtained for the queries and possible ways for visualizing them. We also comment on the benefits and costs of the proposed approach. For explanatory purposes, we use the simple goal diagram depicted in Figure 1. We remark that on this small model it is easy to answer to the questions of the iAnalyst with a direct inspection of the model. On larger models, the added value of using the fAnalyst services becomes more and more relevant.

3.1 Example 1. Assessing and Refining the Informal Model

The iAnalyst assesses the model against possible inconsistencies or redundancies or critical elements, and eventually refines the informal model on the basis of the answer of the fAnalyst to his/her questions.

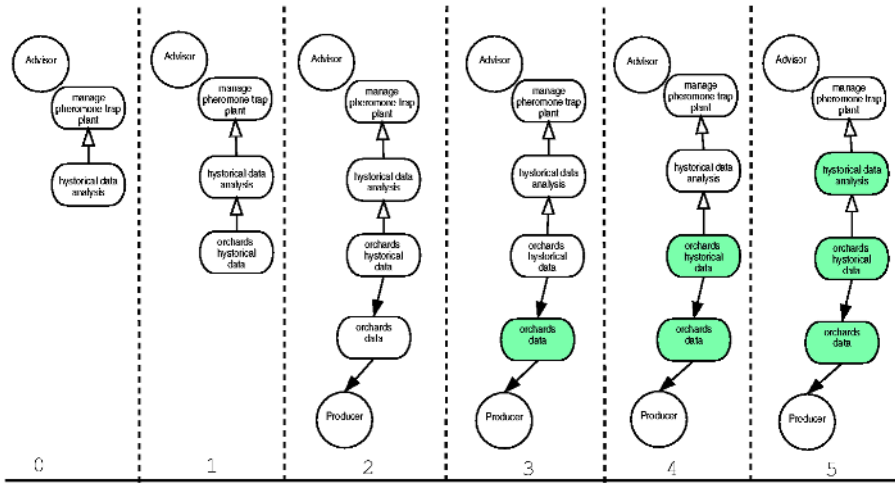


Fig. 3. A scenario where the historical data analysis goal is fulfilled.

Given the goal diagram depicted in Figure 1 a possible question of the iAnalyst can be:

Is it possible for the advisor to perform historical data analysis?

We can formulate this question as the following query:

FULFILLABLE HistoricalDataAnalysis

This query corresponds to the the low level *FT* property:

Global Possibility

$$F (\exists a : \text{Advisor} (\exists hda : \text{HistoricalDataAnalysis} (hda.\text{actor} = a \wedge \text{Fulfilled}(hda))))$$

The *FT* specification admits a scenario that conforms with the above formula, and this scenario can be depicted in terms of a frame sequence, as in Figure 3.

In the scenario, historical data analysis is fulfilled using orchard historical data received from the Producer. Each frame corresponds to a step of the scenario instantiation process: the first frame shows the creation of the goal the query refers to, the subsequent frames complete the creation of sub-goals till a goal delegation is reached (frame 2); in the following frames, the effects of the satisfaction of the delegated goal, shown by the dashed texture, is propagated back, along the goal decomposition till the goal under inspection. The above question can be considered a specific instance of the general question:

Focusing on a specific actor goal, does the current model allow to achieve it in some valid scenario?

If the answer to such questions is positive, the result can be effectively shown by a diagram analogous to the one depicted in Figure 3. If a scenario does not exist, a warning message is emitted.

Another type of question the iAnalyst can pose when analyzing the advisor goal diagram is:

How critical is the dependency on the actor Producer in order to satisfy the main advisor's goal manage pheromone trap plant?

This question can be reformulated as “*Is it possible to fulfill manage pheromone trap plant without fulfilling all the dependencies with the Producer actor?*” This question can be mapped to the following query expressed in the high level query language:

NONCRITICAL Producer FOR ManagePheromoneTrapPlant

which corresponds to the the low level query:

Global Possibility
F ($\exists a : \text{Advisor} ($
 $\exists \text{mftp} : \text{ManagePheromoneTrapPlant} (\text{mftp.actor} = a \wedge \text{Fulfilled}(\text{mftp}) \wedge$
 $\forall p : \text{Producer} (\forall \text{od} : \text{OrchardData} (\text{od.depender} = a \wedge \text{od.dependee} = p \rightarrow$
 $\neg \text{Fulfilled}(\text{od}))))))$

When this query is submitted to the T-TOOL, a witness scenario is generated that conforms to the specification (see Figure 4).

The above question can be generalized by the following:

Focusing on a specific actor, what are the critical dependencies on external actors? That is, what actors can prevent the achievement of a main goal if they do not achieve delegated goals?

This kind of questions can be seen as queries on the graph corresponding to a goal diagram. They are particularly useful when dealing with a complex model for which a direct inspection of the diagram is impractical. For these questions, the informal queries can be automatically translated into the relative *FT* specification, so no additional effort to the iAnalyst is required.

3.2 Example 2. Validating the Informal Model with the Stakeholders

The iAnalyst looks for relevant validation cases to be proposed to the stakeholders in order to drive the validation process and to end up with an agreed model. Validation cases can be suggested by the fAnalyst on the basis of the model structure.

Given the actor diagram depicted in Figure 1 a possible question of the iAnalyst in this case is:

How does the advisor usually operate? Is that he/she always satisfies the goal area collection after satisfying all its sub-goals, according to the following sequence: area analysis via GIS, plant plan and finally dispenser report and order?

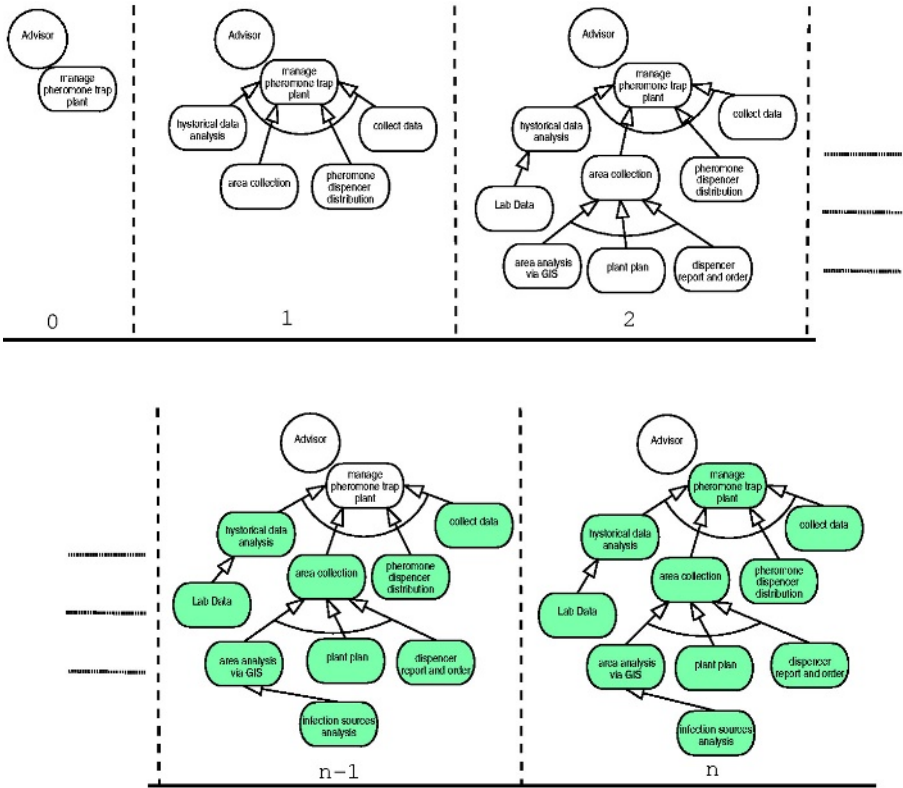


Fig. 4. A scenario where the manage pheromone trap plant goal is fulfilled without any dependency from a Producer.

We can formulate this query as follows:

FULLFILL AreaAnalysisViaGIS **THEN** PlantPlan **THEN** DispenserReportAndOrder **THEN** AreaCollection

This corresponds to the the low level query:

Global Assertion F (

$$\begin{aligned} &\forall a : \text{Advisor} (\forall ac : \text{AreaCollection} (ac.actor = a \rightarrow \\ &\quad \forall aavg : \text{AreaAnalysisViaGIS} (aavg.ac = ac \rightarrow \\ &\quad \quad \forall pp : \text{PlantPlan} (pp.ac = ac \rightarrow \\ &\quad \quad \quad \forall drao : \text{DispenserReportAndOrder} (drao.ac = ac \rightarrow \\ &\quad \quad \quad \quad (\mathbf{Fulfilled}(ac) \wedge \mathbf{Fulfilled}(aavg) \wedge \mathbf{Fulfilled}(pp) \wedge \mathbf{Fulfilled}(drao)) \rightarrow \\ &\quad \quad \quad \quad \mathbf{P} (\mathbf{JustFulfilled}(aavg) \wedge \mathbf{X F} (\mathbf{JustFulfilled}(pp) \wedge \\ &\quad \quad \quad \quad \quad \mathbf{X F} (\mathbf{JustFulfilled}(drao) \wedge \mathbf{X F} \mathbf{JustFulfilled}(ac)))))))))) \end{aligned}$$

When this question is submitted to the T-TOOL, a scenario that violates the assertion is found.

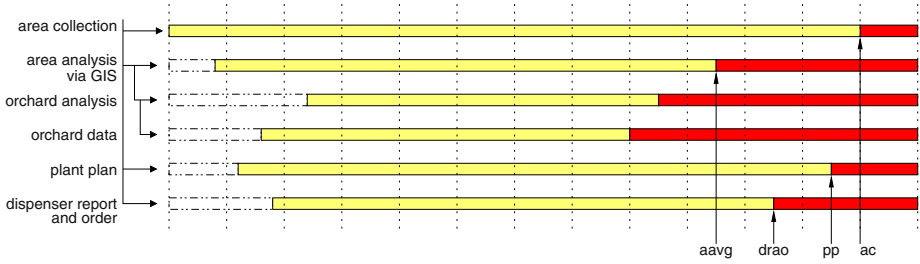


Fig. 5. A validating scenario where the goal area collection is fulfilled.

Figure 5 illustrates this scenario with a bar-charts diagram that shows the ordering in which the goals involved in the specification are created (beginning of the light bar) and fulfilled (beginning of the dark bar). The scenario shows a case where goal dispenser report and order is achieved before plant plan. This scenario can be discussed with the stakeholder, in order to understand whether this order of achievement is possible in the application domain. If it is possible, then the query can be refined in order to take into account that dispenser report and order and plant plan can be achieved in an arbitrary order. If it is not possible, then a temporal constraint between these two goals has to be added in the *FT* model, so that the invalid behavior is discarded. The above question can be considered a specific instance of the general question:

Focusing on a specific actor, is there an implicit temporal ordering in a goal decomposition, that should be explicitied?

This service requires to store all possible ordering of the goals under consideration (a set of bar-chart diagrams like the one of Figure 5 can be used to this purpose) and to annotate those orders that the stakeholder has considered valid.

Another specific question that can be asked for when performing the analysis considered in this second example, is:

In the case more than one plant plan activities can be performed before a dispenser report and order, is it always the case that all instances of plant plan relative to the same orchard have to be fulfilled before fulfilling the dispenser report and order for the orchard?

We can formulate this query as follows:

FULFILL ALL PlantPlan BEFORE DispenserReportAndOrder

which maps to the *FT* assertion:

Global Assertion F (
 $\forall a : \text{Advisor} (\forall ac : \text{AreaCollection} (ac.actor = a \rightarrow$
 $\forall drao : \text{DispenserReportAndOrder} (drao.ac = ac \rightarrow$
 $\text{Justfulfilled}(ac) \rightarrow \forall pp : \text{PlantPlan} (pp.ac = ac \rightarrow \text{Fulfilled}(pp))))))$

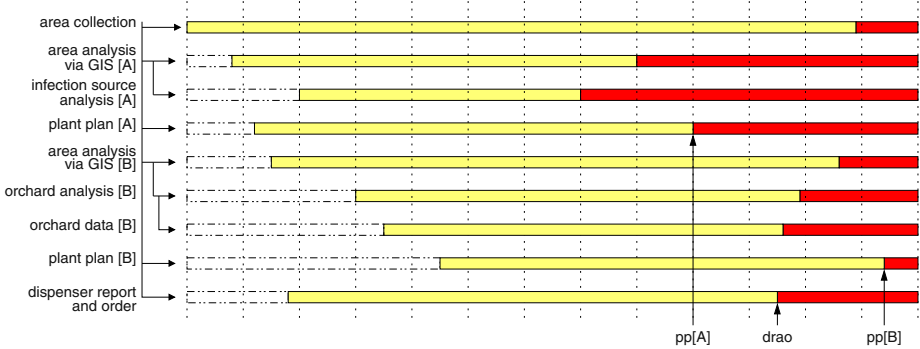


Fig. 6. Another validation scenario for the goal area collection with two instances.

Also in this case, the T-TOOL generates a counter-example scenario, illustrated by the bar-charts diagrams depicted in Figure 6. The scenario shows that it is possible to fulfill the second instance of *plant plan* after the *dispenser report and order* has already been fulfilled. The question can be considered a particular case of the more general one:

Focusing on a specific actor, is there an implicit cardinality on the relationships which models a given goal decomposition that should be explicitated? How do these relationships impact in the temporal ordering of the goal decomposition?

The formalization effort for the queries on implicit temporal orderings and on implicit cardinalities is higher than that of the basic queries seen for Example 1. Indeed, annotations have to be added to the *FT* model in order to implement the constraints on the temporal orders that are agreed with the stakeholder. On the other hand, the understanding of these constraints is very important for a deep understanding of the whole application domain, and their elicitation is very difficult in a purely informal framework.

3.3 Example 3. Managing the Model Evolution

The iAnalyst can exploit the fAnalyst services also to manage the model evolution, catching the creation of inconsistency due to a modification of a (previously consistent) model. In particular, question of the following type could be posed by the iAnalyst:

We remove infection sources analysis. Does this lead to critical dependencies from other actors?

This corresponds to general questions on whether a change in the model makes it impossible to achieve a goal that was previously possible to achieve, or if it introduces new critical goals or dependencies. This question can be solved by

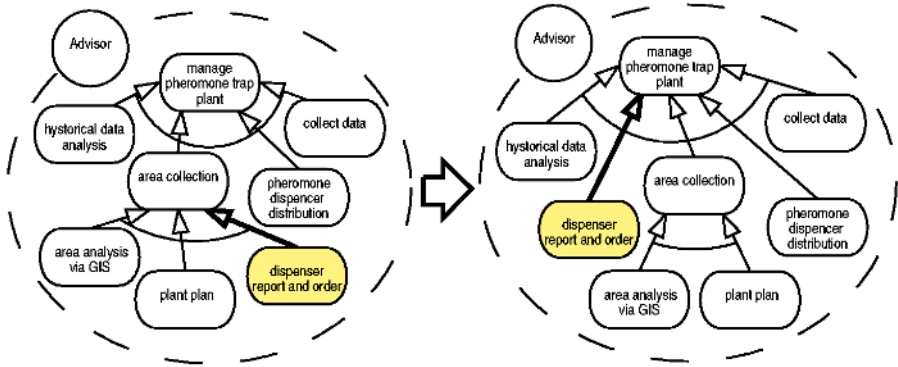


Fig. 7. The modified goal diagram if temporal ordering constraints are taken into account.

re-executing the queries performed in previous steps. If some of the queries fail, then the result produced can be used to correct the specification. If they all succeed, then we can formulate additional queries to validate the specification as described previously.

Assume that in area collection we mark that plant plan has to be satisfied prior to dispenser report and order (as it is the case after a previous query). We now restructure the model, moving goal dispenser report and order one level up (see Figure 7). A second question could be:

Is the temporal order previously described for the goals plant plan and dispenser report and order still respected in the new model?

The question can be seen as a request of advise on possible violation of previously validated temporal constraints, when some change in the AND/OR structure of goals has been introduced. In the specific example, the reconfiguration of the model has invalidated the constraint on the temporal order of the two goals, and a counter-example scenario is returned by the T-TOOL. In order to guarantee that plant plan is satisfied prior to dispenser report and order, a new temporal constraint has to be added to the model, for instance between goals area collection and dispenser report and order.

We remark that the possibility of re-running the queries when the model changes is one of the most important benefits of the approach that we are proposing in the paper. Indeed, it allows to identify the effects of the changes to the properties that the model is supposed to satisfy as soon as the changes are performed on the model.

4 Related Work

The potential advantages of adopting agent-oriented approaches for building complex distributed software systems have been discussed in [13,19]. Current

Agent-Oriented Software Engineering (AOSE) methodologies, such as Gaia [22], MaSE [7], Prometheus [18] and AUML [2] propose to use informal modeling for requirements analysis and system design, even if the importance of Formal Methods is widely recognized in the AOSE community. See [4] for a presentation of some of these techniques and of their application to the specification, to the verification, as well as to the automated generation of an implementation of agent systems. Most of these approaches, however, require a heavy formalization effort and strong skills in order to be used effectively.

Our aim is to allow for a lightweight usage of formal verification techniques, that are used as services in an “informal” development methodology. A recent methodology, called ATOS [14], adopts a similar approach with regard to the detailed design of interaction protocols in Multi-Agent System design. ATOS introduces a textual notation of AUML that can be translated to an extended finite state machine which can be processed by a model checker. ATOS has been exploited to perform formal verification of AUML sequence diagram specification of interaction protocols of Multi-Agent System.

More specifically, in this paper we focus on requirements engineering activities that have been deeply analyzed in [21] which points out also the potential benefits that can be achieved adopting a formal specification approach (even if costs in development and assessment are still considered high). We agree with the following general consideration borrowed from this work, which states that: “the by-products of a formal specification process are often more important than the formal specification itself, including a better informal specification, obtained by feedback from formal expression, structuring and analysis”.

There are several works that propose the application of formal analysis techniques to requirements specifications, but that are outside the frame of AOSE. A discussion of these works is out of the scope of this paper. The interested reader can find this discussion in [10], where *FT* is compared to *KAOS* [16], a framework that supports requirements analysis adopting a goal-oriented approach, and to the *Topoi diagrams* [17].

Some comments are in order on the verification engine used in our framework, namely the NUSMV model checker. The advantage of model checking w.r.t. other verification techniques, (e.g., theorem proving — see [4] for a deeper comparison) is that it allows for an automated verification. This is a fundamental requirements in order to use the formal techniques as “services” of an informal methodology. NUSMV [6] is a flexible model checker that implements several state-of-the-art verification techniques and that provides an open architecture for an easy integration of new algorithms. In the paper we exploited only the NUSMV algorithms for checking linear-time logic (LTL) properties. However, we foresee the possible applications of other NUSMV functionalities (most notably model simulation and the verification of branching time logics like CTL) to answer to new kinds of queries that the iAnalyst may want to formulate.

5 Conclusion and Future Work

This paper described a lightweight usage of formal verification techniques when performing conceptual modeling within an agent-oriented methodology which provides a modeling language that can be used both to build an informal specification or a formal one [3,10].

A preliminary analysis of the proposed framework has been discussed with reference to a set of activities, relevant for requirements engineering, such as requirements elicitation and refinement, user validation of requirements specification, or management of requirements evolution. We considered the decision making process of the analyst when performing those activities and we discussed how it can be supported by formal verification services. Along this line we are defining additional verification services. Moreover, this approach will be extended to other activities in the software development process, such as architectural and detailed design. The proposed framework need to be validated in a systematic way in order to demonstrate its benefits, for instance with respect to the approach that rests on informal modeling only. An ultimate objective, beside that of providing the automatic translation of an informal model to a formal one, is that of developing a tool that supports the analyst and the designer which use informal modeling, for performing the deductive reasoning on a formal specification, by formulating queries analogous to those discussed in the examples of this paper.

References

1. S. W. Ambler. Agile modeling essays, 2003. <http://www.agilemodeling.com/essays.htm>.
2. B. Bauer, J. P. Müller, and J. Odell. Agent UML: A formalism for specifying multiagent software systems. *Int. Journal of Software Engineering and Knowledge Engineering*, 11(3):207–230, 2001.
3. P. Bresciani, P. Giorgini, F. Giunchiglia, J. Mylopoulos, and A. Perini. Tropos: An Agent-Oriented Software Development Methodology. *Journal of Autonomous Agent and Multi-Agent Systems*, 2003. To appear.
4. P. Ciancarini and M. Wooldridge. Agent-oriented software engineering: the State of the Art. In *Agent-Oriented Software Engineering, First International Workshop, AOSE 2000*, number 1957 in LNCS, pages 1–28, Limerick, Ireland, June 2000.
5. P. Ciancarini and M. Wooldridge, editors. *Agent-Oriented Software Engineering*, volume 1957 of *Lecture Notes in AI*. Springer-Verlag, March 2001.
6. A. Cimatti, E. M. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An opensource tool for symbolic model checking. In *Computer Aided Verification*, number 2404 in LNCS, Copenhagen (DK), July 2002. Springer.
7. S. A. Deloach. Analysis and Design using MaSE and agentTool. In *12th Midwest Artificial Intelligence and Cognitive Science Conference (MAICS 2001)*, Miami University, Oxford, Ohio, March 31 - April 1 2001.
8. R. Fuentes, J. J. Gómez-Sanz, and J. Pavón. Activity Theory for the Analysis and Design of Multi-Agent Systems. In *Agent-Oriented Software Engineering, Fourth International Workshop, AOSE 2003*, LNCS, Melbourne, Australia, July 2003.

9. A. Fuxman. Formal analysis of early requirements specifications. Master's thesis, University of Toronto, 2001.
10. A. Fuxman, L. Liu, M. Pistore, M. Roveri, and J. Mylopoulos. Specifying and analyzing early requirements: Some experimental results. In *IEEE Int. Symposium on Requirements Engineering*, Monterey (USA), September 2003. IEEE Computer Society.
11. A. Fuxman, M. Pistore, J. Mylopoulos, and P. Traverso. Model checking early requirements specifications in Tropos. In *IEEE Int. Symposium on Requirements Engineering*, pages 174–181, Toronto (CA), August 2001. IEEE Computer Society.
12. F. Giunchiglia, J. Odell, and G. Weiß, editors. *Agent-Oriented Software Engineering III*. LNCS. Springer-Verlag, Bologna, Italy, Third International Workshop, AOSE2002 edition, July 2002.
13. N. R. Jennings. An Agent-Based approach for building complex software systems. *Communication of the ACM*, April 2001.
14. J. L. Koning and I. Romero-Hernandez. Generating machine processable representations of textual representations of auml. In *Agent-Oriented Software Engineering III, Third International Workshop, AOSE 2002*, number 2585 in LNCS, pages 126–137. Springer, July 2002.
15. P. Kruchten. *The Rational Unified Process: An Introduction*. Addison-Wesley, 2nd edition, 2000.
16. E. Leiter. *Reasoning about Agents in Goal-oriented Requirements Engineering*. PhD thesis, Université Catholique de Louvain, 2001.
17. T. Menzies, J. Powell, and M. E. Houle. Fast formal analysis of requirements via "topoi diagrams". In *the 23rd Int. Conference on Software Engineering*, pages 391–400, Toronto, CA, May 2001. ACM Press.
18. L. Padgham and M. Winikoff. Prometheus: A Methodology for Developing Intelligent Agents. In Giunchiglia et al. [12].
19. A. Perini, P. Bresciani, F. Giunchiglia, P. Giorgini, and J. Mylopoulos. A Knowledge Level Software Engineering Methodology for Agent Oriented Programming. In *Proceedings of Agents 2001*, Montreal, CA, May 2001. ACM.
20. A. Perini and A. Susi. Designing a Decision Support System for Integrated Production in Agriculture. An Agent-Oriented approach. *Environmental Modelling and Software Journal*, 2003. to appear.
21. A. van Lamsweerde. Formal specification: a roadmap. In *ICSE 2000, 22nd International Conference on Software Engineering, Future of Software Engineering Track*, pages 147–159, Limerick Ireland, June 2000. ACM.
22. M. Wooldridge, N. R. Jennings, and D. Kinny. The Gaia Methodology for Agent-Oriented Analysis and Design. *Journal of Autonomous Agents and Multi-Agent Systems*, 3(3), 2000.
23. M. Wooldridge, G. Weiß, and P. Ciancarini, editors. *Agent-Oriented Software Engineering II*. LNCS 2222. Springer-Verlag, Montreal, Canada, Second International Workshop, AOSE2001 edition, May 2001.

The ROADMAP Meta-model for Intelligent Adaptive Multi-agent Systems in Open Environments

Thomas Juan and Leon Sterling

Intelligent Agent Lab, Department of Computer Science and Software Engineering,
University of Melbourne, Victoria 3010, Australia
{tlj, leon}@cs.mu.oz.au
<http://www.cs.mu.oz.au/agentlab>

Abstract. In this paper, we introduce the ROADMAP meta-model, designed to describe intelligent adaptive systems in open environments, using agent concepts such as roles. Developing intelligent adaptive systems creates new challenges in engineering software quality attributes such as correctness and reliability. The ROADMAP meta-model captures our understanding of properties of intelligent adaptive systems and our perspective on organizational concepts such as roles. The meta-model does not solve specific engineering problems, but provides a clean high-level structure where engineering issues can be grouped and classified. Infrastructure to support these issues can then be put in place progressively with consistency. An informal evaluation of the meta-model and comparison to related work is also presented. We expect developers of AOSE methodologies, tools, programming languages and frameworks to benefit from understanding the design and structure of the ROADMAP meta-model. By adopting the meta-model, the resulting methodologies, tools and languages may inherit its desirable characteristics and better support the development of intelligent adaptive systems in open environments.

1 Introduction

In the last decade, the focus of agent research has shifted from single agent systems to multi-agent systems, with emphasis on creating and using artificial organizations of agents [3, 4, 5, 6, 10, 11, 13, 16]. A significant part of the research effort is on exploring different meanings for organizational concepts such as roles. Few attempts have been made to formalize the theory of agent organization as meta-models [4, 13]. However, these attempts did not discuss nor address issues arising from intelligent adaptive systems embedded in open environments.

Intelligent adaptive systems capable of handling open environments have significant commercial value for industry, and have been the focus of extensive research in academia. Such systems present new challenges for engineering traditional quality attributes such as reliability, and inspire new quality attributes such as privacy. As a result, they are difficult to develop using conventional methods.

As intelligent adaptive systems are potentially the most important application domains for agent technologies; we believe the properties and characteristics of such systems must be taken into consideration during the development of AOSE methodologies, tools, programming languages and frameworks. A cost-effective way to simplify the development of intelligent adaptive systems is to build supporting mechanisms directly into the fundamental constructs of AOSE. The ROADMAP meta-model was designed to describe such systems. It defines key constructs for multi-agent systems, such as agents and roles, and their inter-relationships, such as aggregation, in a way that facilitates the development of intelligent adaptive systems. The meta-model formalizes and summarizes our understanding of properties of intelligent adaptive systems in open environments and our perspective on concepts such as roles.

We expect developers of AOSE methodologies, tools, programming languages and frameworks to benefit from understanding the design of the meta-model. By adopting the meta-model, the resulting methodologies, tools and languages may inherit its desirable characteristics and better support the development for intelligent adaptive multi-agent systems. Section 2 presents our analysis of intelligent adaptive systems in open environments from the perspective of engineering software quality attributes [14]. Section 3 introduces the ROADMAP meta-model. Section 4 describes two example applications of the ROADMAP meta-model, namely the ROADMAP methodology [6], and an approach to create custom project-specific methodologies by reusing AOSE features [8, 9]. Section 5 evaluates the ROADMAP meta-model and related work with a set of criteria. Section 6 concludes.

2 Future Agent Systems and the Challenges on Software Quality

In this section, we present our analysis of intelligent adaptive systems in open environments, influenced by our software engineering training, including the need to engineer software quality attributes [14]. We examine how properties of such systems give new meanings to traditional software quality attributes such as correctness and performance. We suggest a class of new and imprecise quality attributes, such as privacy and politeness, based on existing work on engineering non-functional requirements and “soft-goals” in software systems [22,23,24], and explore issues in engineering these new quality attributes.

2.1 Intelligent Adaptive Systems in Open Environments

An intelligent system acts *rationaly* in situations and takes the optimal action to pursue its goals [21]. Performing rational action may require large amount of knowledge and reasoning. For example, for an intelligent search engine to return the most relevant webpages, it must reason about the nature of the user and the context of the search. It must also reason about the content of the webpages it indexed to determine what makes a page relevant.

An adaptive system senses changes in its usage and its environment, and alters its behavior at runtime for better results [25]. For example, a business system may change its structure (architecture) to mirror changes in the human organization. Machine learning could be used by the system for self-optimization.

In an open environment, potentially malicious agents may enter the system at runtime. Therefore we cannot trust all agents to act according to the system-wide goals. Instead, we must create mechanisms to ensure correct system-wide behaviour even when individual agents are malicious and misbehave.

Future multi-agent systems may be expected to have many of these properties. Indeed, IBM's new research initiative codenamed "Autonomic Computing" [17] makes an early attempt to address most of the issues described above.

2.2 Impact on Software Quality

Intelligent adaptive systems in open environments create new challenges in software development. For such systems, current development techniques cannot guarantee quality attributes such as correctness to hold after deployment. For example, correctness is traditionally assured by testing the system before release, against documented requirements. The assurance provided by this approach is lost if the system behaviour changes due to continuous adaptation or environment change, or if new agents in the open environment misbehave. Following the same logic, system performance, reliability, security, usability and maintainability can be compromised due to adaptation or environmental changes. Without explicit representation of system requirements and constant validation at runtime, there is no guarantee that the system functions correctly. This view is shared by earlier work in [12].

To solve the problem, it seems necessary to include constructs to explicitly represent the correct system behaviour and the required level of quality attributes to allow runtime validation of the system. The construct should also allow the desired direction of adaptations to be specified, ensuring that adaptation produces positive and desirable outcomes.

Furthermore, as we rely more on intelligent systems for decision support, new and imprecise quality attributes such as privacy, politeness and benevolence begin to emerge. The exact meanings of these quality attributes, such as good taste and privacy, depend closely on the actual user of the system and the context of each use. Whether the system fulfils the quality attributes is very open to user interpretation and perception. For example, if I rely on assistant agents to suggest gifts to buy for various social occasions, I may expect the choices of the gifts to be legal, appropriate, and show good taste. Yet no general and precise definitions exist for these quality attributes. In another example, we may expect our personal assistant agents to have close knowledge of our daily routines, habits and preferences. Yet we also expect user privacy and do not wish such knowledge to become public. The meaning of privacy and the level of privacy needed by each user are subjective and usually different.

For many intelligent systems, the sheer complexity of their tasks renders it difficult to fully define and test for correctness. To address this issue, we suggest that the

AOSE paradigm must make available constructs to define such quality attributes, in a flexible manner so the definitions can be easily customized for each user at runtime.

The separation of knowledge from hard-coded functionalities allows better re-use and maintenance. It also allows the possibility of engineering quality attributes at the knowledge level. For example, a self-optimizing system can learn the usage patterns and change its internal logic for better performance. By improving the agent's knowledge on analyzing usage patterns and the knowledge on optimizing its own internal logic, we can indirectly improve the system performance significantly.

From the example, we see the potential to improve quality attributes at the knowledge level, in addition to traditional quality engineering at the functionality level. This approach is particularly significant to agent systems as agent oriented programming is at a higher level of abstraction, and in many cases can be viewed as at the knowledge level. It is most natural to perform knowledge level quality engineering in agent systems.

Currently we are not aware of any development methodologies that formally address and systematically support the engineering of quality attributes at the knowledge level. We wish to promote this perspective as essential to developing intelligent adaptive systems and invite new research into this area.

2.3 Summary

Quality requirements of intelligent adaptive systems are dynamic, complex and fuzzy. To facilitate development of these systems, we see the need to provide support for engineering quality attributes, for the entire software development lifecycle in methodologies, CASE tools, programming languages and into the actual applications. As a first step, we support the abstract representation of system functionalities and quality attributes in the constructs of a meta-model for AOSE. We extend existing work [22,23,24] in the context of agent systems, at the level of fundamental constructs for AOSE, and enable runtime reasoning of functionalities and quality attributes.

The meta-model does not aim to provide solutions to every application specific engineering problems such as performance enhancement. Rather, it provides a clear high-level structure under which engineering issues can be grouped, classified and accommodated. At this stage, we take a less formal approach to the meta-model and are not very concerned with the formal semantics of the constructs. We aim to progressively introduce infrastructure for addressing the engineering issues into the meta-model. By adopting the meta-model, methodologies, tools, programming languages and frameworks should consequently provide support for engineering quality attributes consistently.

3 The ROADMAP Meta-model

The ROADMAP meta-model is a generic meta-model for describing multi-agent systems. The ROADMAP meta-model has been derived from the ROADMAP meth-

odology [6] and our earlier work [7]. The meta-model is special in the sense that it describes runtime systems and all constructs shown in the meta-model have concrete physical runtime manifestations. Each concrete runtime construct is instantiated from its respective development-time class. However, we omit the development-time classes from the meta-model for simplicity.

3.1 The Meta-model

Following the analogy that putting people together does not form efficient organizations unless sufficient processes, regulations, infrastructure and organization goals are also in place, we propose to model multi-agent systems by two hierarchies at runtime (see Figure 1).

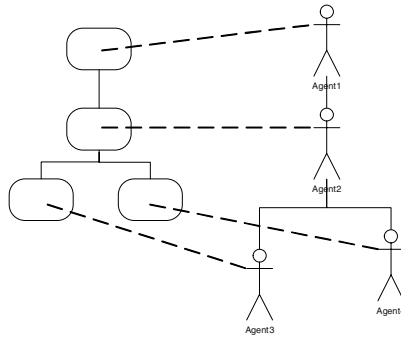
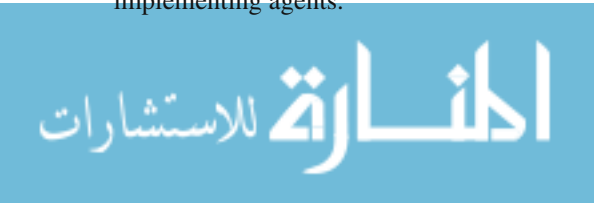


Fig. 1. A multi-agent system viewed as independent hierarchies of agents and roles

The role hierarchy represents a high-level abstract specification of requirements, capturing organizational structures, regulations, processes, goals, responsibilities and various permissions for the agents to function in the system. The agent hierarchy provides concrete implementation of system functionalities.

The ROADMAP meta-model is shown in Figure 2. All the entities in the meta-model are realized at runtime. A role can aggregate other roles, interact with other roles and modify other roles given the proper authorization. A role in the ROADMAP meta-model is defined as in the ROADMAP methodology [6], with two improvements. Figure 3 shows the partial definition of a role.

First, the lifecycle of a role can be roughly divided into states in its liveness responsibility, such as Work in the example below, and further into protocols. We now allow other role attributes to be specified temporally with the states and protocols, by introducing keywords *before*, *during* and *after*. For example, we can restrict the permission to access socket only to the Work state (see Figure 3). Similarly, the Connect protocol can be invoked only if the safety condition “ActiveConnection () < ConnectionPool” holds before the call, as a pre-condition to the protocol (see Figure 3). The use of these keywords gives us new and fine-grained control over the execution of implementing agents.



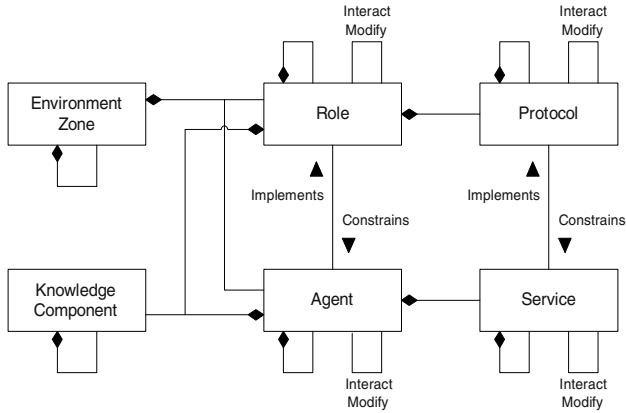


Fig. 2. The ROADMAP meta-model

The second improvement is the use of evaluation functions in roles and the introduction of the Goal attribute. Instead of safety conditions that must hold true in the ROADMAP methodology, functions like Reliability () can now return any value and can be used as evaluation functions for agent performance. These evaluation functions serve as the official measure of quality attributes in the organization. For example, reliability can be defined in many ways. However, for agents taking this particular role, reliability is defined and measured by the Reliability function. Therefore the agent can simply adapt to maximize or minimize the result of this function, according to its Goal attribute. Similarly, a goal of a role may be maximizing privacy, according to an attached evaluation function, say Privacy (), that exactly defines the meaning of privacy for the implementing agents. The keyword “according to” is used to nominate a prioritizing function. The function should return relative importance of active goals given a state or a protocol. The result of this function allows agents to understand the official priority of goals. Agents can decide how to handle conflicting goals and how to spend their resources to achieve the goals accordingly. Some quality goals will have no precise evaluation function. In that case, the best alternative available can be used and the system will probably not perform optimally. Official communications and messages in the organization should go through roles. This ensures agent behaviours can be validated at runtime for correctness and other quality attributes, by invoking the evaluation functions. The functions in role definitions are simply references to implementations that may be in the same role, other roles, an agent or an object in the environment. For example, the function Socket.ResponseTime () in Figure 3 is implemented in an external object named Socket.

Role name:	NetworkTransport
Liveness Responsibility:	NetworkTransport = (Work Wait) ^w Work = Connect . Transmit ^w . Disconnect
Safety Responsibility:	Reliability () > 8 during Work Throughput () > 3 during Transmit ActiveConnection () < ConnectionPool before Connect Socket.ResponseTime () < 4 during Connect
Goals:	Max. Reliability () during Work Max. Throughput () during Transmit Min. ActiveConnection () during Work According to Prioritize ()
Permissions:	Access Sockets during Work
Protocols:	Connect, Transmit and Disconnect and Wait

Fig. 3. Example (partial) definition of a role

An agent is defined as a runtime entity that has a unique identity, communicates using asynchronous and synchronous messages, maintains a list of roles it takes, and maintains a list of currently active roles. The use of messages to model all agent interaction reduces the coupling between agents, as messages can be rejected, forwarded to other agents, logged or played back. An agent can aggregate other agents, and interact with other agents. When agents interact directly without going through roles, the interaction is considered private and does not have the same official status within the organization. For certain organizations private interaction can be undesirable and prohibited.

Figure 4 shows an example of official interaction in an organization between Agent A and Agent B. The message from Agent A is first sent to and validated by its role. If all constraints are satisfied, the message propagates to Agent B’s role. After the message is validated, Agent B receives the message and can now respond to it. As part of the organizational arrangement, the message is also forwarded to Agent C’s role, and to Agent C after validation for monitoring purpose.

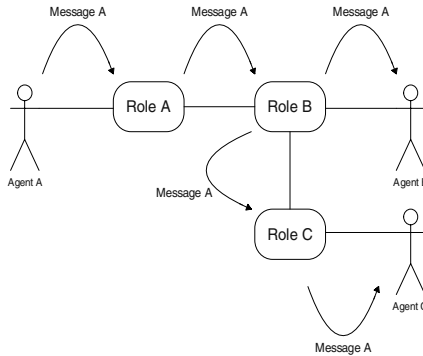


Fig. 4. Successful message passing and forwarding between agents and roles

If the message fails to satisfy constraints from any roles concerned, the message will be rejected and actions will be taken to handle the error. This mechanism ensures that the interaction respects the perspectives of all roles involved. In addition, quantitative results may be produced by evaluation functions within the roles. Such results provide indications to agents on how well their interaction satisfies system requirements on functionalities and quality attributes. This mechanism is somewhat similar to the fitness function in a genetic algorithm [1]. However, it is structured in a modular fashion while allowing different agent implementation architecture. In another word, the roles form an architecture-independent common platform for expressing quality attributes, such as privacy. In theory, it is possible to define the organization, and populate it with learning agents with random initial behaviour. With sufficient training the agents should learn from their roles in the organization to perform correct system functionalities without human intervention. Developers may now design and implement the system quickly without committing to many design decisions, and allow the design decisions to be made by agents at runtime according to the actual usage. Related work on roles can be found in [18,19,20].

A service is a coherent and reusable block of system functionality. A service may include other services for re-use. A protocol of a role constrains the runtime execution of a service of the implementing agent. It is an abstract specification of a service, and contains information such as pre/post conditions and invariants of the service. Fine-grained control over agent execution is achieved by constraining the services agents provide with protocols from roles they take. The recursive nature of roles, agents, protocols and services ensures system scalability. A protocol can be activated to monitor the execution of a service every time the service is run, or sample the service execution randomly on a given interval.

A knowledge component is a modular unit of knowledge. It can aggregate other knowledge components, allowing knowledge scalability and reusability in the system. A knowledge component can be included into a role or an agent, allowing the knowledge sharing, distribution or reuse in the system to be represented and modified at runtime.

Agents and roles are embedded in environment zones, shown as aggregation in the meta-model (see Fig. 2). The environment zones serve two functions in the multi-agent system. It provides uniform non-discriminatory constraints on all agents, for example, gravity applies to all human beings. It also provide infrastructure to facilitate agent services and hence simplify the internal design of agents.

3.2 Development-Time Classes

The ROADMAP meta-model describes runtime systems with concrete constructs that are instantiated at runtime. During development-time, respective classes are created according to the same structure and relationships. For example, the runtime role is instantiated from the development time role class, while runtime agents are instantiated from the development-time agent classes. Figure 5 shows the instantiation relationship.

We focus on the runtime constructs as agent systems are inherently dynamic and the runtime behaviour is of more interest to us.

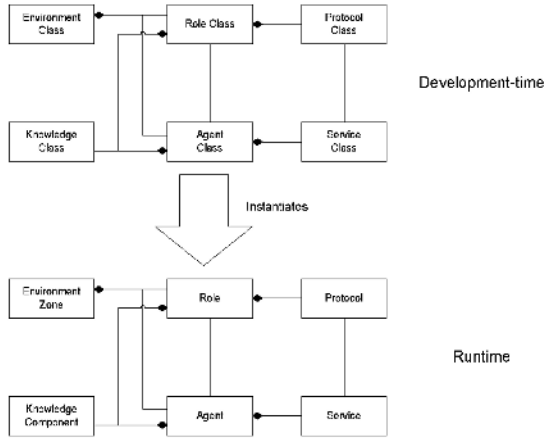


Fig. 5. Constructs in the meta-model are instantiated from their respective development-time classes

4 Example Applications of the ROADMAP Meta-model

In this section we present two example applications of the ROADMAP meta-model to illustrate its applicability and the potential benefit for adopting the meta-model.

4.1 The ROADMAP Methodology

The ROADMAP methodology [6] is designed to support the development of complex open systems. It extends the Gaia methodology [15] with formal environment and knowledge models, and a dynamic role hierarchy to constrain behaviour of agents in the organization. The meta-model was formulated when we attempted to isolate the key concepts in ROADMAP that enable the development of open systems. The key concepts are considered general enough to be useful to other researchers and the sharing of such knowledge is the motivation of this paper.

Figure 6 shows the revised structure of the ROADMAP models. The models are grouped into three categories. The environment model and the knowledge model contain reusable high-level domain information. The use-case model, interaction model, role model, agent model and acquaintance model are application specific. The protocol model and service models describe potentially reusable low level software components.

The ROADMAP methodology closely implements the ROADMAP meta-model. The meta-model can be mapped one-to-one directly onto the shaded models without re-arrangement, as the shaded models contain development-time classes to instantiate constructs in the meta-model (as shown in Figure 5).

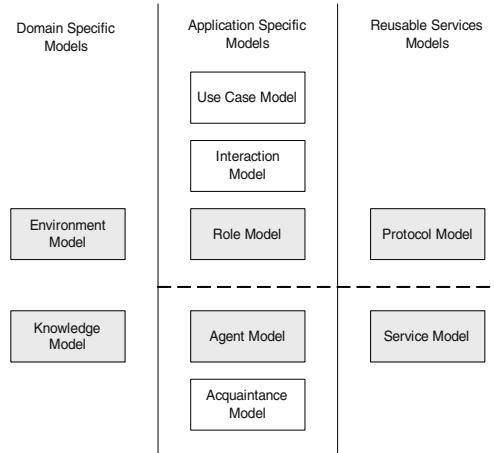


Fig. 6. The models within the ROADMAP methodology

By conforming to the meta-model, the ROADMAP methodology inherits its desirable characteristics and is suitable for developing open, intelligent and adaptive systems. This example shows the applicability of the ROADMAP meta-model.

4.2 Custom AOSE Methodologies by Reusing AOSE Features

In our earlier work [8,9], we described an approach to create reusable modular AOSE features by isolating general-purpose common features from existing AOSE methodologies. The remaining parts of the methodologies are then componentized into special purpose "value-adding" features. This approach empowers the developer to assemble a methodology tailored to the given project by putting appropriate AOSE features together, much like developers building applications from third party off-the-shelf components.

The above approach allows the re-use of methodology features. We envisage that the methodology becomes a living artifact of the project, and changes according to project requirements. Methodology features encapsulate techniques, models, CASE tools and development knowledge such as design patterns. Methodology features can be created to handle particular new quality attributes such as privacy, and be deployed into a project when appropriated.

The benefit of the above approach is obvious. Instead of creating incompatible techniques, models and CASE tools for each methodology, modular and reusable solutions can be created once, and shared within different methodologies. With this approach, specialized features, such as support for safety in medical applications, is more accessible to developers than possible before. This represents significant saving in development cost and learning cost, investment.

The ROADMAP meta-model can be used to ensure semantic consistency between methodology features. By conforming to the same meta-model, the methodology features share the same basic definition for agent constructs such as agents and roles. We expect this to improve interoperability and enables features originated from different methodologies to work together without conflicts.

A role in the ROADMAP meta-model is sufficiently flexible and expressive for AOSE features to base their quality engineering activities on. It is a suitable construct for expressing various quality attributes addressed by different AOSE features, without committing to any agent architecture.

It is worth noting that specific quality attributes are not always required throughout the entire system. For example, politeness may only be required at the user interface, while privacy is required for a set of agents that can access the user's personal information. As a general observation, for the entire system, functionalities need to be modeled and developed. For localized parts of the system, quality attributes may need to be modeled and engineered. The roles in the ROADMAP meta-model allow runtime representation of functional requirements throughout the entire system, and also allow quality attributes to be represented on a localized basis. This allows precise application of engineering effort and avoids paying unnecessary engineering overhead to other parts of the system.

5 Informal Evaluation and Related Work

A meta-model has the potential to ensure consistency between various methodologies, CASE tools, programming languages and frameworks. Indeed, the meta-model for the OO paradigm [2], often known as the Object Model, plays a unifying role for OOSE. Consistent support covering all aspects of software development is a key prerequisite for the acceptance of a software paradigm by mainstream industry practitioners.

By adopting the meta-model, the conforming methodologies, tools and languages will inherit the strengths and weaknesses of the meta-model. It is therefore important to validate any meta-model carefully. In this section, we informally evaluate the ROADMAP meta-model using eight criteria. Related work on the AALADIN meta-model is also evaluated using the criteria.

5.1 Evaluation Criteria

The proposed evaluation criteria are:

1. Runtime scalability of the system
2. Runtime representation of requirements and quality attributes
3. Abstraction of knowledge from functionalities
4. Modularization of knowledge
5. Variable level of agent characteristics at runtime
6. Variable level of quality attributes at runtime
7. Simplicity
8. Ease of learning.

We explain each in turn. Note that we are not claiming that this is a complete set of criteria to evaluate meta-models.

Runtime scalability of the system: The purpose of AOSE is to simplify the development of industry strength applications. Therefore a meta-model should promote scalability of systems to accommodate the application complexity. Adaptive systems in open environments should be able to scale up and down at runtime depending on the usage.

Runtime representation of requirements and quality attributes: As discussed in Section 2, requirements and quality attributes should be represented at runtime, allowing agent behaviour to be validated within the organization. After behaviour changes in adaptive systems, the correctness of the system and other quality attributes are still assured, if agent behaviour complies with the changed roles. Furthermore, such specification (or representation) of requirements and quality attributes should be clearly separated from the implementation.

Abstraction of Knowledge from Functionalities: A meta-model should clearly separate agent knowledge from low-level functionalities. Consequently methodologies, tools and languages implementing the meta-model should discourage developers from hard-wiring knowledge into functional code. The knowledge and the functionalities can then be developed, reused, and maintained separately, at a much lower cost.

A meta-model adhering to this principle also facilitates engineering of quality attributes at the knowledge level.

Modularization of Knowledge: Knowledge should be developed and maintained in modular units with high cohesion and low coupling. The modular approach localizes potential faults and enables easy sharing and re-use of knowledge.

Variable Levels of Agent characteristic at Runtime: We expect a meta-model not to place arbitrary constraints on the level of agent characteristics such as level of intelligence, autonomy, pro-activeness and reactiveness. For example, when it is

desirable, agents in the system should be allowed to become much more autonomous at runtime, without compromising the overall system integrity.

Variable Levels of Quality Attributes at Runtime: In addition to representing quality attributes at runtime, a meta-model should allow the nature and level of quality attributes to be changed at runtime. For example, when hacker attacks are identified, security at relevant parts of the system should be tightened. If the attacks threaten the reliability of the system, reliability may be nominated as a new quality attribute and agents in the system should start working to achieve it.

Simplicity: A meta-model should not be unnecessarily complex, so the resulting methodologies, tools and languages don't inherit the complexity and unnecessarily burden the development of systems.

Ease of Learning: For better acceptance of AOSE, a meta-model should be easy to learn and understand by industry practitioners. In addition to being simple, the meta-model should also be similar to existing approaches such as OOSE, so the user can leverage their existing knowledge for easy transition.

5.2 Evaluation of the ROADMAP Meta-model

The ROADMAP meta-model is scalable at runtime, as all runtime entities can recursively aggregate themselves. The ability for roles and agents to modify other roles and agents allow the system to scale up and down at runtime.

As detailed in Section 3, ROADMAP roles can represent the system requirements and quality attributes, with various role attributes. Having separate hierarchies for roles and agents ensures clear separation between the specification and the implementation of the system. The presence of knowledge component in the meta-model, and the ability to aggregate other knowledge components, indicates knowledge is abstracted away from functionalities (agent services) and managed in a modular hierarchy.

The goals, evaluation functions and safety responsibilities are flexible enough to encode different levels of agent characteristic and quality attributes. The ability to modify roles at runtime ensures no arbitrary constraints are set on levels of agent characteristics and quality attributes in the system at runtime.

The meta-model is very similar to the OO approach as accessing agents through roles is similar to accessing objects through interfaces, and constraining services with protocols is similar to constraining object methods with function signatures in interfaces. The meta-model is considered close to OO and easy for developers to learn and understand. The similarity with OO suggests the meta-model is almost as simple as OO and should not impose unnecessary complexity and burden during development.

The ROADMAP meta-model fulfils these evaluation criteria well.

5.3 Evaluation of Related Works

AALAADIN [4] is a well known meta-model for multi-agent systems based on the concept of agents, groups and roles. Figure 7 shows the structure of AALAADIN.

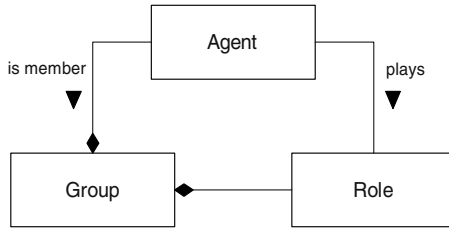


Fig. 7. Structure of the AALAADIN meta-model

AALAADIN agents are atomic and cannot aggregate other agents or groups, while AALAADIN roles are not known to aggregate in anyway. This restricts scalability of systems. The AALAADIN model was then extended in [13], allowing an agent to be atomic or a group (see Figure 8). The extension improves scalability. However, roles can now be part of a group and therefore part of an agent. The lack of clear separation between specification and implementation is undesirable. Both versions of AALAADIN use roles and groups to capture system requirements. However, without concrete and detailed definition for roles and groups, we cannot determine how effective the representation is.

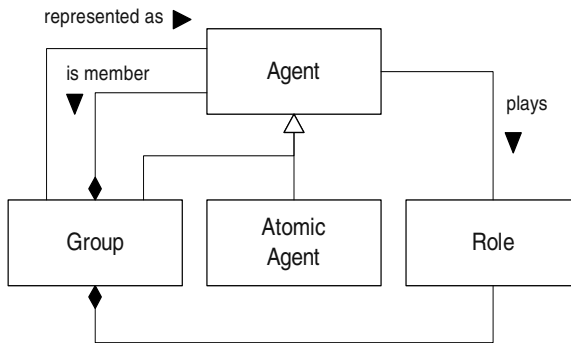


Fig. 8. Structure of the extended AALAADIN meta-model

Knowledge components are not present in either version of AALAADIN, suggesting abstraction of knowledge from functionalities and modularization of knowledge are not supported. Roles are not known to change dynamically at runtime in either version of AALAADIN, implying variable levels of agent characteristic and quality attributes are only possible during development time, not at runtime. Both

meta-models are simple, and although they do not map easily to the OO approach like the ROADMAP meta-model does, they are relatively easy to learn and understand.

6 Conclusion and Future Work

This paper has presented challenges in engineering software quality attributes for intelligent adaptive systems in open environments. We developed a meta-model to handle these challenges. The meta-model does not solve application-specific engineering problems, but provides a clean high-level structure where engineering issues can be grouped, classified and accommodated. Infrastructure to support these issues can then be put in place progressively with consistency. By understanding our design of the ROADMAP meta-model, developers of AOSE methodologies, tools, programming languages and frameworks should gain insight into the challenges in developing intelligent adaptive systems in open environments. By adopting the ROADMAP meta-model, the resulting product may inherit the desired characteristics and better support such systems. We described two applications of the meta-model, and provide some solid initial validation.

In future, we wish to conduct more formal validation and apply the meta-model to other areas of agent research.

Acknowledgements. We like to thank our colleagues at the Intelligent Agent Lab, University of Melbourne and Andrea Omicini for their valuable feedback on this paper. The first author acknowledges support from a part studentship from the Smart Internet Cooperative Research Centre. The second author is partially supported by Discovery Project DP0209297 from the Australian Research Council.

References

1. Beasley, D., An overview of genetic algorithms: Part 1, fundamentals, *University Computing* 15, 58–69, 1993.
2. Booch, G. *Object-Oriented Analysis and Design* (2nd edition). Addison-Wesley: Reading, MA, 1994.
3. Carley, K. and Gasser, L., *Computational Organization Theory, Multiagent System: a modern approach to distributed artificial intelligence*, Weiss, G. (ed). MIT Press, Cambridge, Mass, 1999. 299–330
4. Ferber, J and Gutknecht, O., A meta-model for the analysis and design of organizations in multi-agent systems, *Proc. 3rd Int. Conference on Multi-Agent Systems (ICMAS'98)*, IEEE Computer Society, 1998, 128–135
5. Huhns, M. and Stephens, L., *Multiagent Systems and Societies of Agents, Multiagent System: a modern approach to distributed artificial intelligence*, Weiss, G. (ed). MIT Press, Cambridge, Mass, 1999. 79–120
6. Juan, T., Pearce, A. and Sterling, L., ROADMAP: Extending the Gaia Methodology for Complex Open Systems, *Proceedings of the First International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, p3–10, Bologna, Italy, July 2002.

7. Juan, T. and Sterling, L., A Meta-Model for Intelligent Adaptive Systems in Open Environments (poster), Proc. 2nd Int. Conference on Autonomous Agents and Multi-Agent Systems (AAMAS), Melbourne Australia, July, 2003
8. Juan, T., Sterling, L., Martelli, M. and Mascardi, V., Customizing AOSE Methodologies by Reusing AOSE Features, Proc. 2nd Int. Conference on Autonomous Agents and Multi-Agent Systems (AAMAS), Melbourne Australia, July, 2003
9. Juan, T., Sterling, L. and Winikoff, M., Assembling Agent-Oriented Software Engineering Methodologies from Features, in the Proceedings of the the Third International Workshop on Agent-Oriented Software Engineering, at AAMAS'02, Bologna, Italy, 2002
10. Kendall, E., Agent Software Engineering with Role Modeling, Proc. 1st Int. Workshop on Agent-Oriented Software Engineering, Limerick, Ireland, 2000, 163–170
11. Omicini, A., SODA.: Societies and Infrastructures in the Analysis and Design of Agent-Based Systems, Proc. 1st Int. Workshop on Agent-Oriented Software Engineering, Limerick, Ireland, 2000, 185–194
12. Osterweil, L. and Clarke, L., Continuous Self-Evaluation for the Self-Improvement of Software, in Self-Adaptive Software, Robertson, P., Shrobe, H. and Lagada, R. (eds). 2000, Springer-Verlag: New York, NY. P.27–39
13. Parunak, H.V.D. and Odell, J., Representing Social Structures in UML, Proc. 2st Int. Workshop on Agent-Oriented Software Engineering, Montreal, Canada, 2001, 1–16
14. Pressman, R., Software Engineering: A Practitioner's Approach, 4th edition, McGraw-Hill, 1997
15. Wooldridge, M., Jennings, N. and Kinny, D. The Gaia Methodology for Agent-Oriented Analysis and Design. *Journal of Autonomous Agents and Multi-Agent Systems* 3 (3). 2000, 285–312.
16. Zambonelli, F., Jennings, N. and Wooldridge, M. Organizational Abstractions for the Analysis and Design of Multi-Agent Systems, Proc. 1st Int. Workshop on Agent-Oriented Software Engineering, Limerick, Ireland, 2000, 127–141
17. IBM, Autonomic Computing, <http://www.research.ibm.com/autonomic/>
18. Ferber, J., Gutknecht, O. and Michel, F., From Agents to Organizations: an Organizational View of Multi-Agent Systems, Proc. 4th Int. Workshop on Agent-Oriented Software Engineering, Melbourne, Australia, 2003, in this volume
19. Yan, Q., Mao, X., Zhu, H. and Qi, Z., Modelling Multi-Agent Systems with Soft Genes, Roles and Agents, Proc. 4th Int. Workshop on Agent-Oriented Software Engineering, Melbourne, Australia, 2003, in this volume
20. Odell, J., Van Dyke Parunak, H., Brueckner, S. and Sauter, J., Temporal Aspects of Dynamic Role Assignment, Proc. 4th Int. Workshop on Agent-Oriented Software Engineering, Melbourne, Australia, 2003, in this volume
21. Russell, S. and Norvig, P., *Artificial intelligence: a modern approach*, Prentice Hall, 1995.
22. Yu, E. and Mylopoulos, J., Understanding "why" in software process modelling, analysis and design, Proc. of 16th Int. Conference on Software Engineering, Sorrento, Italy, May, 1994
23. van Lamsweerde, A., Requirements engineering in the year 00: a research perspective, Proc. of 22nd Int. Conference on Software Engineering, Limerick, Ireland, June, 2000
24. Chung., L (Ed.), Nixon, B., Yu, E., Mylopoulos, J. and Nixon, B., *Non-Functional Requirements in Software Engineering*, Kluwer Academic Publishers, October, 1999
25. Robertson, P., Shrobe, H. and Laddaga, R. (eds.), *Self-adaptive software*, Proc. of 1st int. workshop of self-adaptive systems, (IWSAS'00), Oxford, UK, Springer Verlag, April, 2000

Modeling Deployment and Mobility Issues in Multiagent Systems Using AUML

A. Poggi¹, G. Rimassa¹, P. Turci¹, J. Odell², H. Mouratidis³, and G. Manson³

¹ Dipartimento di Ingegneria dell'Informazione, Università degli Studi di Parma
Parco Area delle Scienze 181A, 43100 Parma, Italy
+39 0521 905708
{poggi,rimassa,turci}@ce.unipr.it

² James Odell Associates, 3646 W. Huron River Drive
Ann Arbor, MI USA 48103
+1 (734) 994-0833
email@jamesodell.com

³ Department of Computer Science, University of Sheffield
Regent Court, 211 Portobello Street, Sheffield, S1 4DP, England
+44 (0) 114 2225 888
{haris,g.manson}@dcs.shef.ac.uk

Abstract. This paper demonstrates how UML can be exploited and extended to model the deployment of a multiagent system at the agent level. This is accomplished by extending the formally-based UML 2.0 metamodel to support the semantics of agents, mobile agents and their supporting platforms. Additionally, the UML-based notation, used to model the deployment of a multiagent system, takes advantage of stereotypes to associate an agent-oriented semantics with the model elements used in the diagram. A primary objective of this paper is to demonstrate that the Agent UML (AUML) deployment diagram can be successfully applied to real-world applications. This AUML work is organized as an activity within the FIPA Modeling Technical Committee.

1 Introduction

Software engineering methodologies were and are essential, especially in industry, to improve software productivity, lowering the costs and ensuring a high level of quality. Most people involved in multi-agent systems research realize that a complete modeling language and a complete agent-based development methodology, that covers all phases and activities of the software lifecycle, are necessary in order to take advantage of the agent paradigm and for agent-based technology to be widely adopted by industry.

The research on agent-oriented software engineering is based on the possibility to model a software system at the agent level of abstraction [6][7][12][15][19]. This level of abstraction considers agents as atomic entities that communicate to implement the functionality of the system. Another promising feature of software

agents is mobility. Mobile agents are software entities that can migrate autonomously throughout a network from host to host. This means they are not bounded to the platform they begin execution. Mobile Agents are emerging as an alternative programming-concept for the development of distributed applications [4][5][14][18]. So far, most of the work on the area of mobile agents has been focusing on the technology itself, and the development of agent frameworks to support mobility.

A fundamental agency trait that is captured and modeled by most AOSE methodologies is the *intentionality* of an agent; an external observer can describe an agent by ascribing goals and intentions to it. When taking such an *intentional stance*, the agent coupling with its environment is described in terms of the agent choosing some states of affair over others and acting on its environment trying to bringing them about. Adopting the intentional stance, mobility becomes more than just an infrastructure service; instead, moving is an especially effective way for an agent to act on its environment (if you cannot change your environment, change environment). While the intentional stance is a quite useful one to take in the requirement capture and design models (the goal concepts drives the requirement engineering process in [12]), we feel that it is less fit when describing the deployment aspect of the system. This is because when modeling deployment, one is more concerned with system manageability and environment description.

As evinced from an analysis of the majority of the proposed languages and methodologies, a great effort is spent in defining new metaphors, symbols and diagrammatic notations. But despite this effort the modeling languages proposed still remain incomplete. In particular, up to now, none of the existing agent oriented modeling languages provide concepts and notations to fully capture the multiagent system deployment. Considering mobility, very little work has taken place in defining concepts and notations to capture and model mobile agents. Mobile agents are a crucial part in most agent-based systems and the lack of models to capture them restricts the usefulness of the existing methodologies. Only recently (2001) work was initiated trying to capture mobile agents during the analysis and design stages of the development. Very preliminarily work has been initiated by introducing some concepts to capture mobility at the MaSE methodology [9].

In this paper we consider Agent UML (AUML) [1][15], a modeling language based on UML. The greatest merit of AUML is to use those artifacts that support the development environment which strictly relate to the nearest antecedent technology: object oriented software development. UML is not a methodology by itself, but a suggested notation to be used within the framework of a methodology. The same is true of AUML. The AUML work is organized as an activity within the FIPA Modeling Technical Committee [10]. Initially, the FIPA AUML work was concerned with agent-based class diagrams and interaction diagrams. Other UML diagrams will similarly be exploited in the future in order to cope with the special requirements of multiagent systems. Additionally, new diagrams might be included in AUML to tackle particular concepts not present in UML.

In this paper, we focus on a new subset of an agent-based UML extension for the specification of the deployment of a multiagent system, using and extending UML deployment diagram. In particular, we aim at showing how UML can be exploited and extended to design the deployment of (mobile) multiagent systems at the agent

level. The presented notation takes advantage of stereotypes to associate an agent-oriented semantic with the model elements involved in the diagram.

The next section deals with the relevance and the peculiarity of the configuration and deployment in multiagent systems. Section three describes the rationale used in defining the AUML deployment diagram, specifying the standard representation for its elements in UML models. In particular the first subsection describes how UML can be extended in order to define complete, accurate and unambiguous representation of multiagent system deployment. The second subsection studies in depth the topic of acquaintance relationship between agents. The fourth section explains the notational representation of the semantic concepts defined in the previous section. Section five illustrates how AUML deployment diagram can be successfully applied to a real-world application, since it supports all practical requirements commonly encountered. Finally, the sixth section concludes with a discussion about AUML deployment diagram features and future work.

2 Multiagent System Configuration and Deployment

The role of the deployment diagram in UML is to express the configuration of run-time processing nodes and the components, processes, and objects that reside on these nodes [16]. In several systems, these aspects are quite evident and a deployment diagram does not add real value to the modeling phase. In these cases it can be useful to produce one a posteriori for documentation completeness. Complex systems with several nodes with significantly different computational responsibilities may benefit from the deployment diagram right from the beginning.

Regarding the MAS, the deployment diagram has to represent hosts (servers, front-ends, etc.), resources, physical agents and their acquaintance graphs, and, depending on the framework used in the implementation, MAS platforms. The deployment diagram is very useful to model highly distributed MAS, that is systems in which it is important to visualize the system's current topology and distribution of components and agents, and to reason about the impact of changes on the topology, such as mobile agent systems. As proposed by Mouratidis et al [14] an approach to capture the concept of mobile agents, is to introduce concepts and notations (or use existing ones) to give answers to questions that arise from the use of mobile agents such as *why* a mobile agent moves from one platform to another, *where* the agent moves to, *when* the agent moves, and *how* it reaches the targeted platform. Thus, regarding mobility the deployment diagram should provide answers to those questions.

An important observation to be made at this point is the difference that exists between the *architecture* of a MAS and its implementation-independent *configuration* at deployment time. The architecture of a MAS is a structure that portrays the different kinds of agents and the relationships among them. The architectural description is studied and fixed when designing the MAS. A configuration is an instantiation of an architecture with a chosen arrangement and an appropriate number of agents. One frozen architecture can lead to several configurations. The configuration is tightly linked to the topology and the context of the place where the MAS is rolled out. The architecture is designed so that the possible configurations cover the different system organizational layouts foreseeable in the context of a

project. Agents can be arranged among various machine configurations in order to more effectively use available processing power and network bandwidth.

The deployment of a multiagent system therefore is driven by:

- The system organizational layout: the structure of the company, etc.
- The network topology: technical environment and its constraints such as the topologies characteristics, the network maps and data rates, data servers location, gateways, firewalls, etc.
- The interests area: where are the stakeholders (users, system managers, providers, etc.)

In order to describe the configuration description of a MAS, at deployment time, UML models must represent agent constructs, capturing their structure and semantics. Since UML predates the agent oriented software engineering, it does not contain model elements that express the structure and semantics of multiagent system. UML was designed to be extensible, however, and provides standard extension mechanisms for defining new model elements. These mechanisms can be used to define new model elements to represent new entities like agents, their acquaintances and MAS platforms. We use those mechanisms in our work to extend the UML deployment diagram enable the modeling of Multiagent systems. We call the extended diagram derived from our work, AUML Deployment diagram.

3 AUML Deployment Diagram Semantics

The aim of this section is to provide complete semantics for all modeling notations used in the AUML deployment diagram. While it may appear lengthy, it is important to be accurate for the sake of clarity.

3.1 Extending UML Metamodel

A major feature of the multiagent system approach to software development is the reliance on the social level of abstraction: this allowed researchers in the MAS field to take inspiration and leverage results from social sciences, where they deal with complex and dynamically changing systems. However, the shift towards a social perspective in multiagent systems should not suggest forgetting the main attributes of the single agent, namely autonomy and situatedness. When inserted into a society, each member agent becomes situated in an hybrid environment, arising partly from social and institutional entities and partly from entities external to the agent society. The diagrammatic representation of a concrete MAS should be able to fully depict this hybrid situatedness, showing agents and their social and natural environment as a whole. The social aspect of agent situatedness can be captured by an oriented graph connecting agents with arcs. This graph is called “acquaintance” graph.

In simple client/server systems, usually it is assumed that clients know the server beforehand but the server does not know a client until it is contacted by it. These strict assumptions, common in multi-tier client/server systems, make the acquaintance graph trivial, which explains why it is generally not included in the system diagrams, but a MAS architecture can result in arbitrary acquaintance graphs, so they have to be explicitly represented.

From the above considerations it follows that even if the deployment diagram is part of the architectural models and its scope is to model the static deployment view of a system, some aspects of MAS deployment diagrams are related to the behavioral models. The AUML deployment diagram therefore must be more expressive than the corresponding reference UML deployment diagram, and must include the acquaintance relationships between agents.

The architecture of UML is based on a four-layer metamodel structure; in our discussion about the semantics of MAS deployment diagram we consider the user objects, model and metamodel layers. Moreover in order to allow an accurate interpretation of the semantics choices, supporting our proposal, we make a clear distinction between a conceptual entity, its instances, its representation or implementation and the instances of its representations.

Having said this, in the following we try to give a precise definition of the terms involved in the deployment diagram, mapping them in UML model elements.

For the following, refer to Figure 1.

We consider a starting point of our dissertation the definition of the model element “agent” as a stereotype of the metaclass Class [3][2]. The stereotype extension mechanism provides a way of defining virtual subclasses of UML metaclasses with new metaattributes and additional semantics.

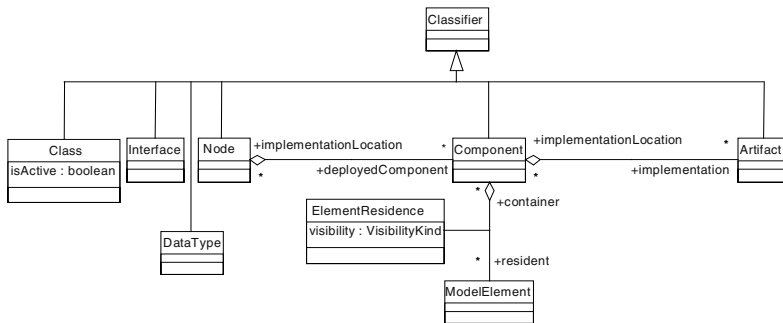


Fig. 1. UML 1.4 Core Package - Classifiers

What is applicable to a metaclass Class is therefore, by definition, applicable to an Agent Class. Agent Class defines a set of elements, that we call “Agents”¹ (instances of the Agent Class), which have the same structural and behavioral characteristics. Moreover Agent Class is a conceptual element declared in an intensional way as a collection of features and inherits participation in Associations. The stereotype «acquaintance» is applied to an Association between Agent Classes to denote that messages may be sent between their instances.

¹ An Agent has at least one thread of control and runs concurrently with other Agents; Agent Class is therefore a subclass of Class with the attribute “isActive” always true.

An Artifact is a concrete element² that we can define with a good approximation as a structured set of bytes. The implementation of an Agent Class can be memorized in one or more Artifacts. Executable Artifacts can be loaded in memory and be associated to one or more executable threads. If an Artifact contains an implementation of an Agent Class, we can say that the copy, in memory, is able to create concrete elements that implement Agents, instances of the Agent Class itself. An Artifact may constitute the implementation of a deployable Component.

At this point a brief digression concerning the definition of the Component model element is necessary, given its fundamental role in the deployment diagram.

3.1.1 Component: Attempt at Clarity

While analyzing the UML Reference Manual and the submissions to UML 2.0 RFP, it seems very difficult to reach an agreement on the definition of the Component model element.

We try to outline how the definition and consequently the role of the Component model element is changed throughout UML 1.3, 1.4, 1.5 and the submissions to UML 2.0 RFP. Finally we aim at giving a satisfactory and functioning definition of the component role inside the AUML model. The scope is to properly define a component and its relationship with files, classes, interfaces and mainly agents in order to provide the correct semantics to the modeling notation used in AUML deployment diagram.

In the UML Reference Manual the component is seen as an element belonging to the implementation domain. However, the Component definition in the UML 1.3 model is not satisfactory. Cris Kobryn [13], co-chair of the UML Revision Task Force, said: "The current semantics for the component construct are vague and overlap the semantics of related classifiers, such as class and subsystem. In order to fully support component-based development, the semantics of components should be refined and the overlap with related constructs should be reduced".

UML 1.4 proposes a revision of the Component definition in which the implementation level is shifted from Component to Artifact. The Component is no longer explicitly defined as a physical piece of implementation and it is no longer the *implementationLocation* of the ModelElements, but only their container. The Component is implemented by one or more Artifacts (see Figure 2); the idea that Artifacts "contain" the implementation of the Component comes to light. ModelElements and Artifacts are not implemented or owned by the Component, but they reside in it. This can be seen as a move towards a vision of the Component as a conceptual element instead of as an element belonging to the implementation domain.

The revision of the definition of the Component element by UML 1.4 (and now UML 1.5), however, has been insufficient to solve the problems brought up for the UML 2.0 version. When examining the third revised version of the OMG RFP submission [17], concerning the UML 2.0 Superstructure³, we can see significant

² In general we use the term "concrete element" to denote an active process, a dynamic library, an instance of an Implementation Agent Class, etc. We use the term "conceptual element" to denote for example an Agent Class.

³ In the following we will use the simplified expression UML 2.0 to denote the revised submission to OMG RFP, concerning the UML 2.0 Superstructure.

changes as far as Component is concerned. In UML 2.0, a Component represents a modular part of a system and its definition is given in terms of provided and required interfaces. Component may be made manifest by one or more Artifacts; between Artifact and Component there is an “implements” relationship. A Node is associated with a deployment of an Artifact and it is also associated with a set of elements, e.g. Components, which are deployed on it. The last one is a derived association in the sense that these elements (e.g. Components) have one or more implementing Artifacts that are deployed on the Node. In UML 2.0 the deployment diagram is the only form of implementation diagram and its role is to show how Artifacts are allocated to Nodes. It is plain that the Component is now no longer used solely at the implementation level; instead it can now be employed *as a* conceptual element, as well.

Another interesting aspect of UML 2.0 is the notion of the *active class*. Active classes are those whose objects “owns a thread and can initiate control activity.” In other words, active objects are very compatible with the definition of agent. Furthermore, since the UML 2.0 metaclass Component is a subtype of Class, then components too inherit the proactive, autonomous nature of the active class.

Bearing this in mind, in this paper we consider Component as a conceptual element. The Component is defined as a container of one or more Artifacts. The property of the Component, as a conceptual element, is the property of being able to host other conceptual elements, like Agent Classes. At the implementation level, the meaning of the role “resident” (characterizing the Artifact) is that the Executable Artifact, binding to the Component, contains code, which is able to create, at the execution time, instances of the [implementation of the] Agent Classes, resident in the Component.

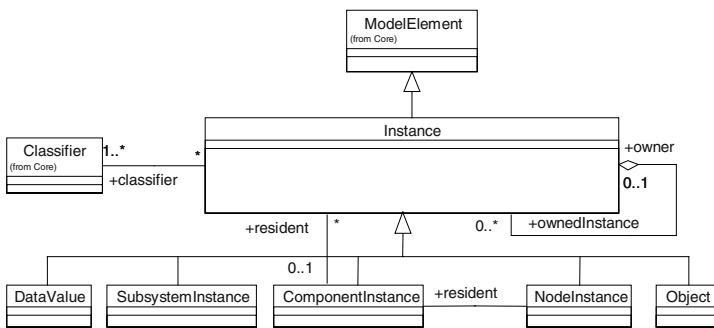


Fig. 2. UML 1.4 Common Behaviour – Instances

The meaning that we usually give to ComponentInstance is an instance of the Executable Artifact associated to the Component itself. So when we speak of ComponentInstance we refer to its executable part. If a Component hosts conceptual elements, like Agent Classes, a ComponentInstance will host concrete elements,



instances of the implementations of these conceptual elements; in this specific case, concrete agents⁴.

A `NodeInstance` is an instance of a `Node`. A collection of `ComponentInstances` may reside on a `NodeInstance`. In the metamodel each `ComponentInstance` that resides on a `NodeInstance` must be an instance of a `Component` that resides on the corresponding `Node` (see Figure 2).

Concerning mobile agents, as it was mentioned, they have the ability to migrate to different nodes within the network. We define the `NodeInstance` they begin execution as their *origin*, and the `NodeInstance` they stop execution, after finish their task, as their *destination*. We define the intermediate path between the origin and the destination as *mobility path*.

In the *AUML deployment diagram* we capture the mobile agents of the system in a static nature (that means these diagrams do not capture dynamics (such as sequence) of the movement). Such a diagram provides notations to capture the mobile agents of the system, along with their *origin*, the *destination*, the nodes they might visit, and the mobile agent's *mobility paths*.

An *AUML deployment diagram* captures mobility if for any link, connected two `NodeInstances`, with the stereotype «moves» the following holds:

- There is a corresponding Mobile Agent (MA)
- The *origin*, *destination* and *mobility path* of the mobile agent are specified.

The stereotype «moves» specifies the movement of the mobile agent from one `NodeInstance` to another, and a note indicates the purpose of the movement. To indicate the *origin* of the mobile agent we introduce the stereotype «home» and we apply this stereotype to the model element `Agent Class`. The stereotype «destination», of the `Agent Class`, is introduced to specify the *destination* of the mobile agent. It must be noticed that *destination* of a mobile agent can be its *origin*. For example when a mobile agent is sent somewhere, obtain some information and returns back to its *origin* to report the results. Finally we introduce the stereotype «visitor» to indicate that the mobile agent visits a node.

3.1.2 Agent Platform: Relating to Middleware

Multiagent systems, as any software development approach, benefit from sound methodology and notation but there is more to them. Concrete issues such as APIs, libraries and infrastructure are also very important to achieve a success when adopting MAS development in a software project. Among all the various UML diagrams, the Component and Deployment Diagrams are the ones most tightly related to concrete software infrastructures and middleware; therefore, when defining the AUML Deployment Diagram, it is natural to consider the agent oriented middleware standards and products available, to try and find useful abstractions to model them within AUML.

FIPA issued a series of agent system specifications that had as their goal interoperable agent systems. In particular this work includes specifications for an abstract architecture, which has, as natural expression, a set of object classes comprising an

⁴ The conceptual element `Agent` denotes an instance that originates from an `Agent Class`. The term “concrete agent” denotes an instance that originates from an implementation of an `Agent Class`

agent platform. An agent platform provides the physical infrastructure in which agents can be deployed. FIPA agents leverage the facilities offered by the agent platform for realizing their functionalities; moreover agent platform provides message transport service and white and yellow page services to external agents resident on other platforms.

We believe that agent middleware, and in particular FIPA compliant agent platforms, have to play a fundamental role in the development of agent based application and that information related to agent platforms has to be reported in the AUML deployment diagram.

Two UML metamodel elements were considered: Subsystem, which seems more appropriate for modeling the internal view of an agent platform and Component, which seems to be the natural choice on the basis of its characteristic of exhibiting services. Moreover, in the UML metamodel a Component is a subclass of Classifier, and as a Classifier it may also have its own Features and realize Interfaces. This assertion of “individuality” of the Component is important since it means the component not only exposes the interfaces of artifacts deployed in it but it can implement them directly.

As far as the revised submission to OMG RFP for the UML 2.0 Superstructure is concerned, the things significantly change. The Component no longer realizes the provided interfaces directly, instead, its parts jointly realize all interfaces offered by it. The Subsystem is a kind of (subclass of) Component and can expose interfaces in terms of elements that are visible from outside. A subsystem’s contained elements have more cohesion and interaction with other element within the same subsystem and less with elements in external subsystem. In this context it seems more suitable to map the agent platform to the Subsystem model element. Since the UML 2.0 specifications still being finalized, the choice falls on the Component, seen more as an infrastructure for agents deployed in it rather than as a mere agent container. But this specification will be revised to take full advantage of the changes made to the UML by the future version.

We therefore define the model element “agentPlatform” as a stereotype of the metaclass Component. It should be noted that the concept of an agent platform does not mean that all agents resident on an agent platform have to be co-located on the same host computer. FIPA envisages a variety of different agent platforms from single processes containing lightweight agent threads, to fully distributed agent platforms built around proprietary or open middleware standards.

One AgentPlatform Component, therefore, can span more than one Node, if the agents belonging to this platform are deployed on more than one Node.

The model element AgentPlatformInstance represents an instance of an AgentPlatform Component and it can host concrete agents⁵.

While giving semantics to AUML Deployment Diagrams, this section also hinted at how wide their domain is. Starting from the abstract classifiers view, they come as far as encompassing actual middleware infrastructures. However, most software projects tend not to include Deployment Diagrams in their deliverables; this is

⁵ An AgentPlatformInstance is a valid AgentPlatform Component instance if every instance in it is a direct instance of some element in the system model, e.g. an Agent.

because most distributed software systems nowadays follow very simple and fixed deployment schemas.

On the other hand, the social stance advocated in multiagent systems modeling increases the importance of the acquaintance relationship between agents. Moreover, beyond the diverse acquaintance structures that can arise from social role modeling, there are even more fundamental reasons to put the system deployment view near the center of the stage when modeling multiagent systems.

Two such reasons are *concreteness* and *situatedness*: they are rooted in the very notion of agency and will be further discussed in the next subsection.

3.2 Concrete and Situated Multiagent Systems

Considering the software engineering process in more detail, promoters of the MAS approach generally stress its suitability for heterogeneous distributed systems. Those systems are exactly the ones where deployment issues can become nontrivial and deserve to be analyzed and addressed with properly designed techniques. In the standard UML 1.4 specifications, the deployment diagram is nothing but the expression of a special static model including Node elements (UML 1.4, Section 3.96.4), and can contain only very generic associations, such as communication between nodes and dependency between components.

On the other hand, there is another UML diagram that nicely combines structural interconnection information with interaction description: the collaboration diagram. A Collaboration is the structure of the participants playing roles in the performance of a specific task, whereas an Interaction is the communication pattern of the Instances playing those same roles (UML 1.4, Section 2.10.1).

Among the several distinguishing properties that have been proposed as agency traits, here we are interested in *concreteness* and *situatedness*. Concreteness is not really an agency trait, but we use it here to compare concrete agents with abstract roles. Using roles and their associations (static social model), along with role interaction specifications (dynamic social model), provides a logical model of a MAS society. However, actual acquaintance occurs between concrete agents, possibly due to them playing a social role but also affected by their physical location and the jurisdiction domains they belong to.

The second property, *situatedness*, is more fundamental. Even in single agent modeling, an agent is always represented as immersed within an *environment* it can interact with through sensing and acting. This means that the agent oriented approach tries to model not just the system but also its boundaries. Mobility further enriches the scenario; when taking the situated agent perspective, an agent perceives its relocation as a sudden change of the environment. Depending on the data space management policies in use, such change can be strongly discontinuous (some resources disappear, others are suddenly available). When moving from a single agent to a whole MAS, we notice that acquaintance is nothing but *social situatedness*: just like we used to model the world outside a single agent describing which resources are known to the agent and which sensing and acting interactions are possible, we now model the society outside an agent describing its acquaintances and the social roles (i.e. the possible conversation patterns) through which interaction occurs.

It is interesting to notice that when an agent acts on the environment it changes it, but generally the change is limited to a small fraction of the whole world. Similarly, when an agent is involved in a conversation, it generally modifies only a small fraction of the society it lives in (e.g. the mental attitudes of the agent he is having a conversation with). Therefore, we can characterize the agent environment (both physical and social) as a *slowly varying process* with respect to the agent internal processes.

We claim that collaboration diagrams, while quite well suited to MAS in general, still are not appropriate to represent a concrete and situated MAS. This is for two reasons, the first one related to concreteness and the second one to situatedness.

Dealing with concreteness, one finds out that the elements appearing in a Collaboration are quite abstract, being roles played by various Classifier and Association elements; this is fine for modeling social structure and interaction at a logical level, but not to express the actual configuration of an agent society, made by concrete agents and not simply by roles. Moreover, the mobility aspect is attached to concrete agents and not to their social roles. Modeling mobility at the role level would mean modeling the transfer of a social responsibility between agents deployed at different locations, without any link to real code mobility.

The second reason is due to the Link element. A collaboration diagram ends up representing Instances conforming to one or more ClassifierRole and Links conforming to an AssociationRole. Messages are exchanged among Instances and travel over Links. We want to model the social situatedness of an agent participating in a MAS, so we would like the link to represent an arc of the acquaintance graph.

A Link is just a support for messages and its lifetime can be very short. Only Links with a lifetime that spans several Interactions can be meaningful arcs of the acquaintance graph. Therefore the model element Acquaintance Link, instance of an Acquaintance Association, denotes a valuable or lasting acquaintance relationship between concrete agents.

4 AUML Deployment Diagram Notation

A deployment diagram is a graph of nodes connected by communication associations. A node in UML deployment diagram is an element that exists at run time and represents physical hosts, that is a computational resource, generally having some memory and processing capability, on which agents may be deployed. Graphically a node is rendered as a cube (see Figure 3). Every node instance has a name, a textual string, and a type. The most common kind of relationships among nodes are associations that represent physical connections. In Figure 4 you see that the nodes have names such as Client and AppServer. These terms are recognizable to the developers within the organization because those are the terms they use on a daily basis.

Generic components are depicted, as they are in UML standard, as rectangle with two smaller rectangles jutting out from the left-hand side. The concrete agents may be contained within the component instance symbols to indicate that the items reside on the component instances. A concrete agent is rendered as rectangle with a name. Two

primary forms of information may be supplied for an agent name: instance, and class. The general form of describing the agent name in AUML is:

```
instance-name : class6
```

An important observation to make is that these diagrams contain concrete agents and not agent classes. This means that a single agent icon can be an instance of one or more agent classes (i.e., many agent roles can be played together by a single concrete agent).

Agents belonging to the same agent platform are grouped together. An agent platform is a kind of Component, indicating which agents are housed on the platform itself. Every agent platform must have a name that distinguishes it from other platforms; a name is a textual string.

Agents are connected to other agents by acquaintance relationships; this indicates that one agent could communicate with the “known” agents by means of interactions protocol. A directed graph is used to show the agent acquaintance graph. The directed graph identifies communication pathways between concrete agents playing the roles involved in an interaction scenario. A non-directed edge denotes that both concrete agents, playing the roles, know each others.

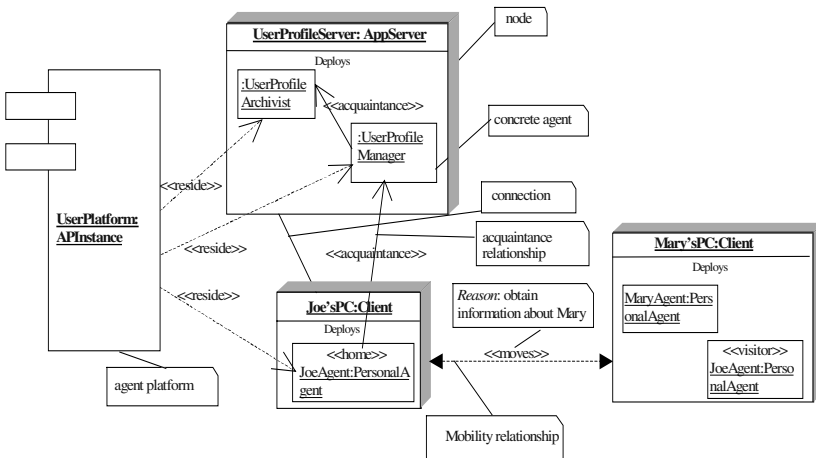


Fig. 3. AUML deployment diagram

The mobile agents are represented with a stereotype that indicates the status of the mobile agent (*home, visitor, destination*). The paths are represented as dashed lines, with the arrows pointing towards the node that the mobile agent moves to. A double arrow (both sides of the path) indicates the mobile agent moves both directions. When a *«destination»* tag is not used, it is assumed by default that the mobile agent returns to its home (*origin*).

⁶ UML 1 also includes the role specification. The difference between the notion of class and role was not well defined. Both defined an entity that possessed structure and behaviour. The only difference between the two was that roles were also involved interactive behavior. In UML 2.0 therefore, classes depicted in interaction diagrams are referred to as “roles.”

5 Applying AUML Deployment Diagram

The main objective of this section is to prove that AUML deployment diagram can be successfully applied to model mobile agents and real-world applications.

We focus our attention on CoMMA [8][11] system, an open, agent-based system for the management of a corporate memory. CoMMA, a FIPA compliant agent system implemented through the use of JADE framework, is the result of an international project funded by European Commission.

CoMMA system uses agents for wrapping information repositories (i.e., the corporate memory), for the retrieval of information, for enhancing scaling, flexibility and extensibility of the corporate memory and to adapt the system interface to the users. These tasks are performed through the cooperation among different kinds of agents that can be divided in four sub-societies: document and annotation management; ontology (enterprise and user models) management; user management and agent interconnection and matchmaking.

The agents from the document dedicated sub-society are concerned with the exploitation of the documents and annotations composing the corporate memory. They will search and retrieve the references matching the query of the user with the help of the ontological agents. The agents from the ontology dedicated sub-society are concerned with the management of the ontological aspects of the information retrieval activity especially the queries about the hierarchy of concepts and the different views. The agents from user dedicated sub-society are concerned with the interface, the monitoring, the assistance and the adaptation to the user. Finally the agents from the interconnection dedicated sub-society are in charge of the matchmaking of the other agents based upon their respective needs.

The deployment diagram can help a lot in focusing on the actual system, because the CoMMA knowledge management solution is made by a highly distributed system deployed over a structured, managed corporate network. Several possible deployment strategies can be envisaged. Between these are: three-tier deployment, characterized by a common database server tier, made by machines shared by the whole corporation, and department based deployment, where the corporate intranet is divided into various departments, and each one of them has its own database servers.

With the aid of the three-tier deployment scenario of the CoMMA system we intend to show the use of the AUML deployment diagram. In addition, we assume that some of the agents used in the system are mobile agents. This assumption helps to demonstrate, in a real-life system, how AUML deployment diagram can be used to model mobile agents.

5.1 CoMMA MAS Three-Tier Deployment Scenario

The Figure 4 depicts the three-tier deployment, so called because the physical computers hosting the agents are divided into clients (*client tier*), application servers (*middle tier*) and database servers (*DB tier*). Such a deployment strategy has the main advantage of matching one of the most popular intranet structures.

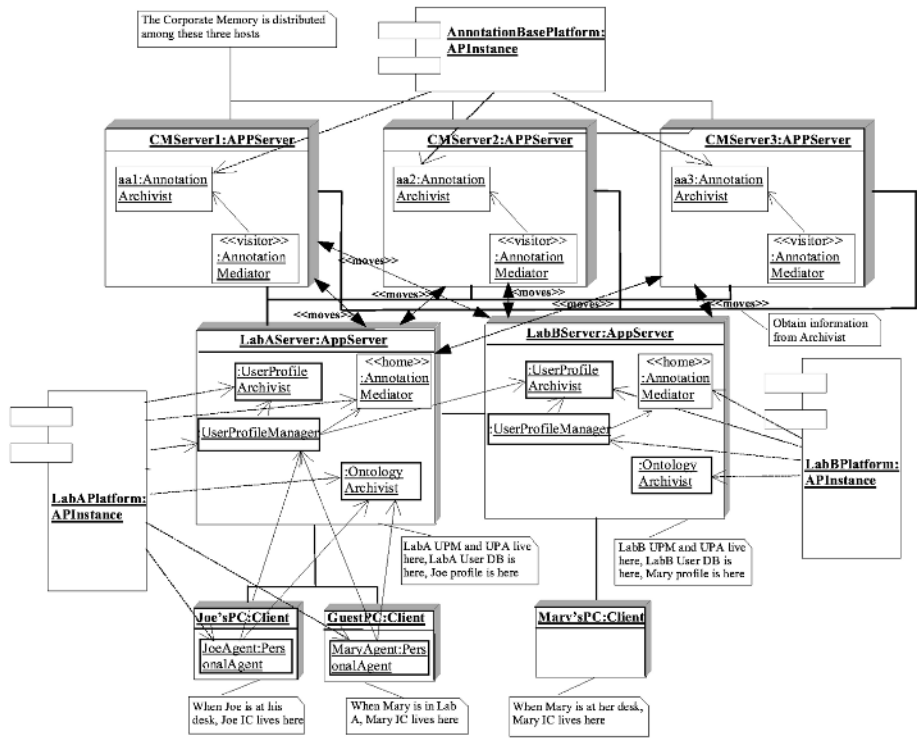


Fig. 4. CoMMA MAS Three-Tier Deployment

The agent acquaintance graph shows the case of two users logged into the system. The first user, named *Joe*, belongs to the *Lab A* organizational unit and is currently sitting in front of his PC. The second one, named *Mary*, belongs to the *Lab B* organizational unit and is currently logged on the *Guest PC* of the *Lab A*, that is not her home location. According to the three-tier architecture, only *Interface Controller* agents live on the client machines (performing pure presentation) and only *Annotation Archivist* agents live on the database servers (performing pure data management). The middle tier, as in classical three-tier architectures, connects the client tier and the data base tier (through *Directory Facilitator* and *Annotation Mediator* agents) and is where the ontology and user model management occurs. In this deployment, the ontology and the user model belong to the middle tier and not to the data base tier because they are supposed to be much smaller in size than the annotation base and they are not distributed.

In this deployment example, there is an agent platform spanning the whole *Lab A*, another platform spanning the *Lab B* and a third platform that contains all the *Annotation Archivist* agents managing the distributed annotation base. This means that any agent within the *Lab A* has an a priori knowledge of the *Directory Facilitator* living on the *Lab A Server* machine, so registering the *User Profile Manager*, the *Ontology Archivist* and the *Annotation Mediator* with the local *Director Facilitator* is enough to connect the user sub-society and the ontology sub-society.

The *Interface Controller* responsible for Mary runs on the *Guest PC* and is acquainted with the *Lab A User Profile Manager*, since *Lab A* is the current location for Mary; the *User Profile Manager* is acquainted with the *Lab B User Profile Archivist*, since *Lab B* is the home location for Mary. The *Annotation Mediator* is a mobile agent; this agent can move to the node where the *Annotation Archivist*, which owns a high percentage of information required, resides.

6 Discussion and Conclusions

In this paper we made a proposal of an AUML deployment diagram, that is an UML deployment diagram enhanced with agent based concepts, and we also described how this kind of diagram can be used to model mobile agents. In addition, we use it to describe the deployment of the CoMMA system

The actual adoption in the CoMMA project of a preliminary version of the notation we are proposing provided some early feedback on the usefulness of an AUML Deployment Diagram. However, more work needs to be done to gather more comments within the AUML proponents and users; an issue that needs to be addressed deals with non-social situatedness. An agent surroundings comprise not only its acquaintances, but also several non-agent entities such as the resources it uses and manages, the events it can perceive, the concrete actions it can perform. As an example, the CoMMA system Deployment Diagram should have shown not only agents and acquaintance links, but also resources (data repositories in the Knowledge Management domain in the case of CoMMA project). Representing non-agent entities and their connections with agents in the AUML Deployment Diagram requires further study to integrate them in the UML metamodel, and it is left as subject for future work.

References

- [1] Agent UML - AUML Home Page. Available at <http://www.auml.org>.
- [2] Bauer B., J. P. Müller, and J. Odell. Agent UML: A Formalism for Specifying Multiagent Interaction. Agent-Oriented Software Engineering, Paolo Ciancarini and Michael Wooldridge eds., Springer, Berlin, pp. 91-103, 2001.
- [3] Bauer B., UML Class Diagrams: Revisited in the Context of Agent-Based Systems, In Proc. of Agent-Oriented Software Engineering (AOSE), pp.1-8, Agents 2001, Montreal
- [4] Brenner W., Z. Rüdiger, and W. Hartmut. Intelligent Software Agents: Foundations and Applications. Springer-Verlag, pp. 55-67, Berlin, 1998
- [5] Caglayan A., C. G. Harrison. Agent Sourcebook: A Complete Guide to Desktop, Internet, and Intranet Agents. John Wiley, 1997
- [6] Caire, G., P. Chainho, R. Evans, F. Garijo, J. Gomez Sanz, P. Kearney, F. Leal, P. Massonet, J. Pavon, and J. Stark. Agent Oriented Analysis using MESSAGE/UML. In Proc. Second International Workshop on Agent-Oriented Software Engineering (AOSE-2001), Montreal, Canada, May, 2001, 101-107.
- [7] Ciancarini P., and M.J. Wooldridge. Agent-Oriented Software Engineering. Lecture Notes in Computer Science, 1957, Springer-Verlag, 2001.

- [8] CoMMA Project Home Page. Available at <http://www.ii.atosgroup.com/sophia/comma/HomePage.htm>.
- [9] Self A.L., A.S. DeLoach. Designing and Specifying Mobility within the Multiagent Systems Engineering Methodology. SAC 2003, Melbourne, Florida, March, 2003.
- [10] FIPA Modeling Technical Committee – Home Page – Available at <http://www.fipa.org/activities/modeling.html>
- [11] Gandon F., A. Poggi, G. Rimassa, and P. Turci. Multi-Agents Corporate Memory Management System. - Applied Artificial Intelligence, 9-10 (22): 699-720, 2002.
- [12] Giunchiglia F., J. Mylopoulos, and A. Perini. The Tropos Software Development Methodology: Processes, Models and Diagrams. In Proc. AAMAS Conference, 2002.
- [13] Kobryn C., Modeling Components and Frameworks with UML, Communications of ACM, Vol. 43 No. 10, October 2000.
- [14] Mouratidis H., J. Odell, and G. Manson. Extending the Unified Modeling Language to Model Mobile Agents. In *OOPSLA 2002 Agent-Oriented Methodologies Workshop*. 2002. Seattle, WA.
- [15] Odell J., H. van Dyke Parunak, and B. Bauer. Extending UML for agents. In G. Wagner, Y. Lesperance, and E. Yu, editors, Proc. of the 2nd Int. Workshop on Agent-Oriented Information Systems, Berlin, 2000. iCue Publishing.
- [16] OMG, UML 2.0 Superstructure RFP, Object Management Group, document ad/00-09-02, issued September 15, 2000.
- [17] OMG, U2 Partners' UML2 Superstructure, 3rd revised submission, Object Management Group, document ad/03-04-01, issued April 18, 2003.
- [18] White J. E. Mobile Agents. Software Agents, Jeffrey Bradshaw ed., MIT Press, Cambridge, MA, 1997, pp. 437-472.
- [19] Wooldridge M., N.R. Jennings, and D. Kinny. The Gaia Methodology for Agent-Oriented Analysis and Design. Kluwer Academic Press, 2000.

A Knowledge-Based Methodology for Designing Reliable Multi-agent Systems

Mark Klein

Massachusetts Institute of Technology
Cambridge MA 02139
(617) 253-6796
m_klein@mit.edu

Abstract. This paper describes a methodology that system designers can use to identify, and find suitable responses for, potential failure modes (henceforth called 'exceptions') in multi-agent systems.

1 Introduction

Multi-agent systems must be able to operate robustly despite many possible failure modes ('exceptions') that can occur. Traditionally, multi-agent system (MAS) designers have largely relied on their experience and intuition in order to anticipate all the ways their systems can fail, and how these problems can best be addressed. While methodologies such as failure mode effects analysis (FMEA) do exist [1], they simply provide a systematic procedure for analyzing systems, without offering specific insights into what exceptions can occur or how they can be resolved.

This approach is becoming untenable, however, as the scale, heterogeneity and openness of multi-agent systems increases. Multi-agent systems, with their promise of self-organized behavior, are being looked to as a way to smoothly and rapidly integrate the activities of large collections of software entities that may never have worked together before. The agents in such 'open' contexts will not have been designed under centralized control, and must operate on the infrastructures at hand. Such systems must be able to operate effectively despite a bewildering range of possible exceptions. We have identified two main classes of exceptions that can occur in MAS contexts:

- ◆ **Commitment Violations:** This category consists of problems where some entities in the MAS do not properly discharge their commitments to each other, e.g. when a subcontractor is overdue with a task, a message is delivered garbled or late, or a host computer crashes. Even the best production code includes an average of 3 design faults per 1000 lines of code [2], and in open systems we can expect a wide range of code quality as well as actively malicious agents.
- ◆ **Emergent Dysfunctions:** This category consist of dysfunctional behaviors that emerge from the locally correct behavior of many agents. There are many examples of such dysfunctions, ranging from social dilemmas such as the

- ◆ ‘tragedy of the commons’ [3], to wild variations in resource utilization [4] [5], and timing artifacts such as ‘resource poaching’ (wherein earlier low priority tasks freeze out later high-priority tasks from access to critical resources) [6]. Such exceptions are especially problematic because they do not represent errors per se, but rather the unexpected consequences of simple coordination mechanisms applied in complex environments.

The challenge of identifying exceptions and their resolutions is complicated by the fact that expertise on this subject is scattered across multiple disciplines that include computer science, industrial engineering, economics, management science, biology, and the complex system sciences. MAS designers are thus unlikely to be cognizant of all the expertise potentially relevant to their tasks.

This paper describes a methodology that multi-agent system (MAS) designers can use to identify, and find suitable responses for, these potential failures (henceforth called ‘exceptions’). We present the core exception analysis methodology in section 2, and then describe (in section 3) how an augmentation of the MIT Process Handbook captures exception handling expertise in a way that can greatly increase the speed and comprehensiveness of exception analysis.

2 Exception Analysis

Our exception analysis methodology is based on the insight that coordination fundamentally involves the making of commitments [7] [8] [9], and that exceptions (i.e. coordination failures) can as a result be viewed as *violations* of the commitments agents require of one another. Exception analysis thus consists of the following steps:

- ◆ Identify the *commitments* agents require of one another
- ◆ Identify the *processes* by which these commitments are achieved
- ◆ Identify the ways these processes can violate these commitments (i.e. the *exceptions*)
- ◆ Identify the ways these exception can be handled (i.e. the exception *handlers*)

We consider these steps in the paragraphs below. To make the discussion more concrete, we will describe them in context of the ‘Contract Net’ (CNET), a well-known auction-based task-sharing mechanism known as the [10].

In this protocol, an agent (the ‘contractor’) identifies a task that it cannot or chooses not to do locally and attempts to find another agent (the ‘subcontractor’) to perform the task. It begins by creating a Request For Bids (RFB) which describes the desired work, and then sending it to potential subcontractors (typically identified using an agent known as a ‘matchmaker’). Interested subcontractors respond with bids (specifying such issues as the price and time needed to perform the task) from which the contractor selects a winner. This is thus a first-price sealed-bid auction. The winning agent, once notified of the award, performs the work (potentially subcontracting out its own subtasks as needed) and submits the results to the contractor.

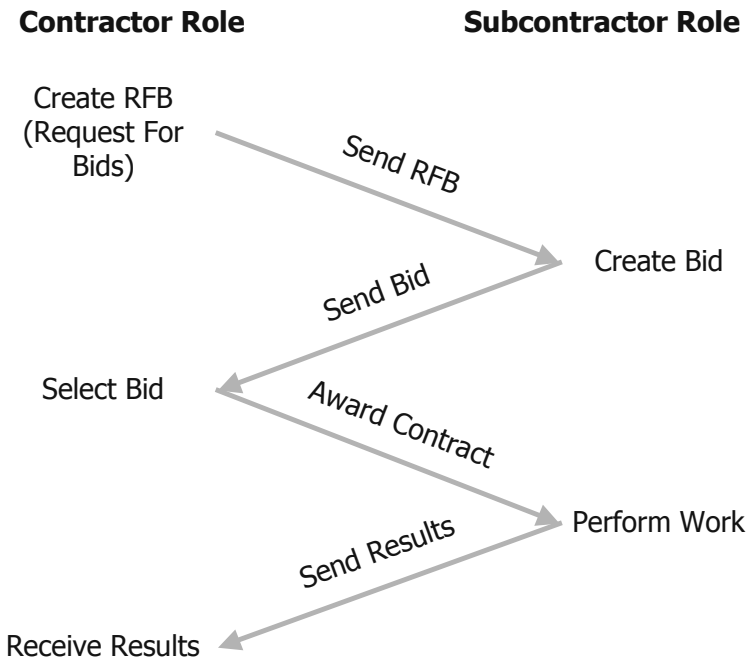


Fig. 1. The Contract Net Coordination Mechanism

We can see that there are thus at least three key agent types in CNET: the contractor, subcontractor, and matchmaker.

2.1 Identifying Commitments and Processes

The commitments involved in a coordination mechanism represent all the places where one agent depends on some other agent to achieve its goals [7]. They can be identified in a straightforward way by linking commitments to the agent processes that achieve them, and linking these processes to the commitments they in turn require to execute successfully. The first commitment in CNET, for example, is the contractors' requirement that it have a list of agents potentially suitable for performing the task T it wishes to subcontract out:

C1: *matchmaker* receives list of subcontractor agents suitable for task T

In CNET, this requirement is discharged by the matchmaking process enacted by the matchmaker agent:

P1: *matchmaker* finds matches for skill set S

In order for process P1 to discharge commitment C1, it in turn requires such other commitments as:

- C2: *matchmaker* receives subcontractor agent skill notifications
- C3: *matchmaker* receives correct skill set S in timely way
- C4: *matchmaker* receives sufficient computational resources to run effectively
- C5: *matchmaker* code was programmed correctly
- C6: *matchmaker* results message is sent quickly and correctly to requesting contractor agent

Commitment C4, in turn, is achieved by a host computer:

- P2: *host computer* provides computational resources for hosted agents

If we follow this process in sufficient depth we can, in principle, exhaustively identify the commitments (and associated processes) involved in a given coordination mechanism.

The explicit identification of all commitments is critical because often exceptions occur because no mechanism was put in place to ensure the satisfaction of some important but overlooked implicit commitment. Also note that commitments should be enumerated *from the idealized perspective of each agent*. For example, in an auction the seller ideally wants the following commitment:

- C6: *seller* receives bids representing true bidder valuation for good

even though in many situations the bidder may have no intention of fulfilling this commitment (e.g. in auctions where true-value revelation is not the dominant strategy). Many important exceptions represent violations of such ideal-case commitments.

2.2 Identifying Commitment Violations

The next step is to identify, for each commitment, how that commitment can be violated by the process selected to achieve it, i.e. what the possible exceptions are. An initial set of exceptions can be identified simply as the logical negations of the commitment itself. For example, the commitment to send a message consists of three components: delivering the *right message* to the *right place* at the *right time*. Any process selected to achieve these commitments thus has three possible failure modes:

- E1: *sender* delivers the wrong message (e.g. the message is garbled)
- E2: *sender* delivers message to the wrong place
- E3: *sender* delivers message at the wrong time (e.g. the message arrives late or never)

Not all exception types, however, can be identified so simply. Process P2 above, for example, can violate its commitments due to exceptions that include:

- E4: *host computer* experiences a denial of service attack
E5: *host computer* is infected by a virus

The range of possible exception types seems to be limited only by human imagination. This introduces a experiential component into exception analysis; one must rely on one's previous experience to identify possible exception types, and an exhaustive identification may not be possible

2.3 Identifying Exception Handlers

Once we have identified the exceptions that potentially characterize a given MAS process, we are ready to identify possible processes for *handling* these exceptions. Exception E1, for example, can be *detected* by the process:

- P3: *sender* performs error-detecting checksum on message contents

and it can be *resolved* by:

- P4: *sender* re-sends message

As with exceptions themselves, the range of possible exception handling processes appears to be limited only by human creativity. Also note that exception handling processes, just like any other MAS process, can of course require their own commitments and face their own exceptions.

This exception analysis procedure is systematic but potentially very time-consuming, and it still requires that MAS designers have a substantial amount of expertise about possible exceptions and how they can be handled, so the possibility of missing important exceptions or valuable exception handling techniques remains.

3 Exploiting a Knowledge Base

This challenge has led us to explore whether it is possible to systematically accumulate exception-related expertise so that designers can benefit from ideas drawn from other designers, and from multiple disciplines, in order to perform exception analysis more quickly and completely. Our approach has been to build upon the MIT Process Handbook, a process knowledge repository which has been under development at the Center for Coordination Science (CCS) for about 10 years [11]. The key concept underlying the Handbook is that processes can be arranged into a taxonomy, with very generic processes at one extreme and increasingly *specialized* processes towards the other. Such taxonomies have two useful properties. One is that attributes of generic entities tend to be inherited by their 'specializations', so one can capture useful generalizations that apply to a wide range of processes. The other is that similar entities (e.g. processes with similar purposes) tend to appear close to one another.

We extended this schema to allow it to capture the results of applying the exception analysis methodology described above. This was accomplished (see [12] [13]) by defining:

1. a taxonomy of commitment types, where commitments can be linked to the processes that require them as well as the processes that achieve them, and
2. a taxonomy of exception types, where exceptions can be linked to the processes they impact as well as the processes appropriate for handling them.

Using this extended schema, we have developed a knowledge base that consists of the results of applying our exception analysis methodology to a range of more or less abstract MAS coordination processes and their component sub-processes. We have also implemented a web-based interface for accessing and editing the contents of this knowledge base. The examples presented in this paper are all drawn from this knowledge base.

A MAS designer can use a knowledge base structured in this way to facilitate exception analysis as follows:

1. Consult the knowledge base to find the generic processes that subsume, or closely match, the processes used in the MAS of interest.
2. Identify which of the exceptions listed for those generic processes in the knowledge base appear to be important for this particular MAS.
3. For each of these exceptions, identify which of the exception handler(s) described in the knowledge base seem best suited for this MAS. These exception handlers should, of course, be submitted to the same exception analysis procedures as the other MAS processes.

The power of this approach comes from the fact that a relatively small corpus of abstract commitments, exceptions and process models is, when represented in this way, capable of capturing a surprisingly high proportion of the exception handling expertise we need. We describe how this works in more detail below.

3.1 Finding the Matching Generic Processes

The taxonomic organization of processes in the Handbook knowledge base makes it straightforward to find matching generic process(es). The procedure is similar to finding a book in a library. One simply traverses down the subject taxonomy from the top, selecting the most appropriate sub-categories at each step, until the desired section is reached.

We have developed a taxonomy of the most widely-used MAS coordination processes, ranging from market mechanisms to distributed planning to game-theoretic and stigmergic approaches (available on-line at <http://franc2.mit.edu/pq1/> with login = guest and password = guest). We have focused our initial efforts on auctions because their wide applicability, scalability, simplicity and well-understood properties make them widely used by MAS designers (see [14] for a description of the taxonomy). Every auction mechanism captured in the Handbook includes a textual description as well as an enumeration of its subtasks, required commitments, and the commitments it achieves.

Using this information, one can readily determine which processes in the taxonomy match the process of interest. Imagine for example that a MAS designer has developed a task allocation scheme wherein subcontractor agents send in sealed bids in order to compete for subtasks. To find a matching generic process, he or she would start at the ‘resource sharing process’ branch of the taxonomy, and traverse down from there, quickly reaching the fragment of the process taxonomy devoted to auction mechanisms:

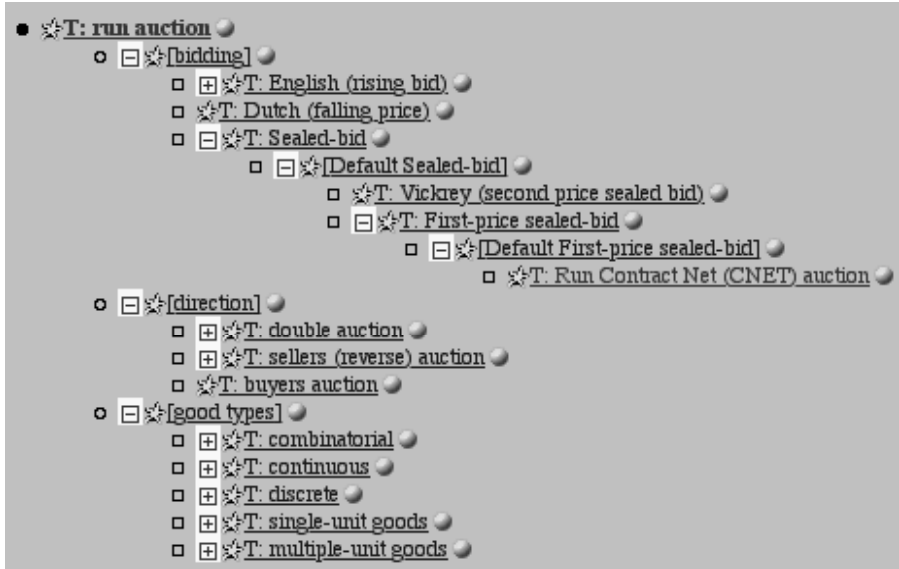


Fig. 2. A portion of the auction mechanism taxonomy

Using this portion of the taxonomy, the designer can quickly determine that his or her task allocation scheme is an instance of the generic Contract Net process described in the knowledge base. In addition to enabling the quick identification of relevant exception types (see below), finding the matching generic processes can increase the completeness of the MAS model: one can look at the components and requirements in the generic process and check whether any are missing in the current MAS model.

3.2 Finding Applicable Exceptions

Once the matching generic processes have been identified, we can then identify the exceptions that the MAS is potentially prone to. This is straightforward because each process in the knowledge base is linked directly to its characteristic exceptions. All auctions, for example, are a specialization of the abstract ‘pull-based sharing’ process, which represents mechanisms wherein resources, e.g. subcontractor agents, are allocated based on consumer requests rather centralized budgeting. If we consult the Handbook knowledge base we find that pull-based sharing is prone to such ‘emergent

dysfunction' exceptions as "resource poaching" and "synchronized jump thrashing" (Figure 3):

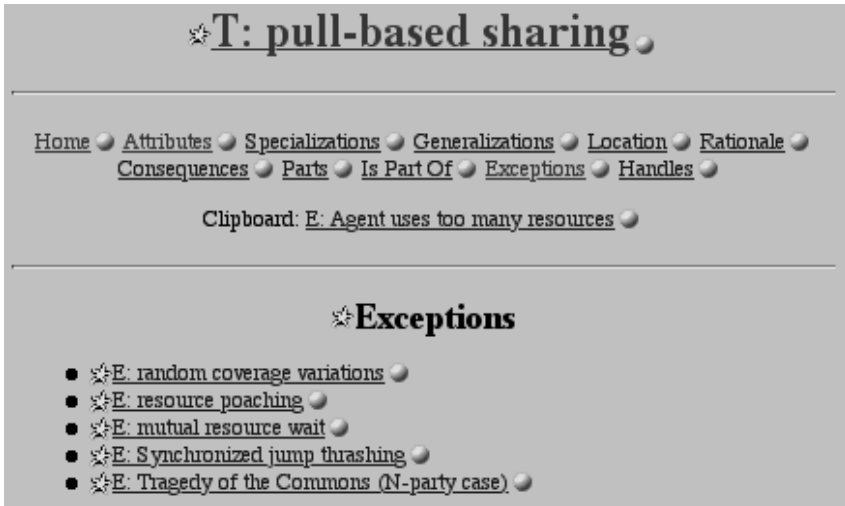


Fig. 3. An example of exceptions for a generic process

The fact that auction mechanisms are prone to such emergent dysfunctions is a potentially powerful and easily missed insight. Auction mechanisms are typically designed by economists interested in equilibrium behavior, and implemented by computer scientists, while the dynamics of resource sharing are studied by researchers in the complex systems field, which grew mainly out of physics. This is an example, therefore, of how a taxonomic approach, based as it is on the identification of powerful generalizations, can foster the cross-disciplinary transfer of insights about exception handling challenges and solutions.

3.3 Finding Applicable Handlers

Once one has determined which exceptions are important for a particular MAS, the next step is to identify the appropriate exception handlers. This is straightforward because exceptions in the Handbook knowledge base are linked directly to the exception handling processes appropriate for them. Our current knowledge base, for example, notes that the exception "Synchronized Jump Thrashing" (where resource consumers generate oscillatory or even chaotic resource utilization behavior due to delayed resource quality information) can be detected using signal processing techniques and resolved by carefully timed modification of resource availability messages [4].

There are four classes of exception handlers in the knowledge base, divided into two pairs. If an exception has not yet occurred, we can use:

- ◆ Exception *anticipation* processes, which uncover situations where a given class of exception is likely to occur. Resource poaching, for example, can be

anticipated when there is a flood of long duration tasks requiring scarce, non-preempting subcontractors to perform them.

- ◆ Exception *avoidance* processes, which reduce or eliminate the likelihood of a given class of exception. Resource poaching can be avoided, for example, by allowing subcontractors to preempt their current tasks in favor of higher priority pending tasks.

If the exception has already occurred, we can use:

- ◆ Exception *detection* processes, which detect when an exception has actually occurred. Some exceptions, such as bidder collusion for example, are difficult to anticipate but can be detected post-hoc by looking at bid price patterns.
- ◆ Exception *resolution* processes, which resolve an exception once it has happened. One resolution for bidder collusion, for example, is to penalize and/or kick out the colluding bidders and re-start the auction for the good in question.

The exceptions in our knowledge base are arranged, like processes, into a taxonomic structure (Figure 4):

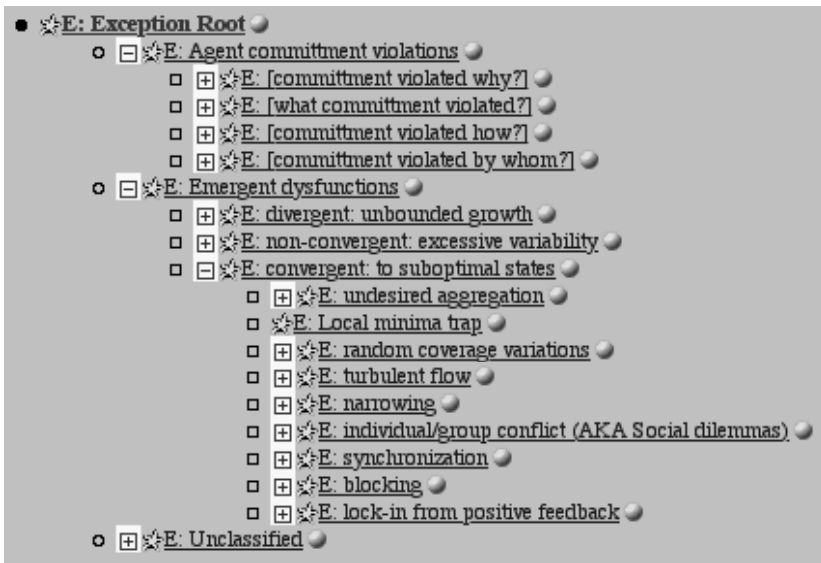


Fig. 4. A fragment of the exception taxonomy

Exceptions are grouped into classes that share similar underlying causes and thus similar exception handling techniques. This implies that when a new exception handling process is entered into the knowledge base, it can be placed in a way that makes explicit the full range of exceptions to which it is applicable.

Where multiple alternative handler processes exist for addressing a particular exception, the Handbook knowledge base allows one to describe the pros and cons of the handlers using tradeoff tables. The taxonomic structure of the knowledge base

also facilitates the design of innovative handler processes through re-combining elements of existing handlers [15].

Whatever handlers we select can themselves be made subject to the exception analysis approach described above in order to further increase the robustness of the MAS. Reputation mechanisms, for example, have been put forth as handlers for many classes of agent commitment violation exceptions. Our knowledge base captures the fact that such mechanisms can be sabotaged by such exceptions as dishonest ratings.

4 Contributions

Limited space has only allowed us to sketch out the knowledge-based exception analysis approach we have been developing. We hope, however, that the key benefits have been made clear. Existing exception analysis techniques leave the identification of possible exceptions and handlers up to the MAS designers themselves. The knowledge-based exception analysis procedure we describe, by contrast, makes it possible to systematically enumerate all the points where exceptions can occur, quickly identify what exceptions may appear at those points, and suggest how they may be handled. Because this knowledge base represents the accumulation of expertise drawn from many different designers and disciplines, it has the potential of identifying exception types and handler techniques that may otherwise be overlooked by the MAS designer.

We have applied our tools and knowledge base to exception analysis in a range of domains including futures trading, multi-agent system task allocation, and concurrent engineering. Our preliminary assessment is that the methodology can be effective in helping designers design more reliable multi-agent systems.

5 Future Work

We are continuing to accumulate a knowledge base of exception types and handlers, currently focusing on market-based sharing, collaborative synthesis, and emergent dynamical dysfunctions. We are also developing software agents that use this knowledge base to do real-time exception detection and intervention in multi-agent systems. For additional information about this and related work, see <http://ccs.mit.edu/klein/>.

Acknowledgements. This work was supported by the DARPA CoABS program as well as the NSF Computational and Social Systems program. The author gratefully acknowledges the important contributions made by Chrysanthos Dellarocas, George Herman, Thomas Malone, Simon Parsons, John Quimby, Juan-Antonio Rodriguez-Aguilar, and others.

References

- [1] Hunt, J., D.R. Pugh, and C.J. Price (1995). *Failure mode effects analysis: a practical application of functional modeling*. Applied Artificial Intelligence. **9**(1): p. 33–44.
- [2] Gray, J. and A. Reuter (1993). *Transaction Processing : Concepts and Techniques*. Morgan Kaufmann series in data management systems. San Mateo, Calif. USA: Morgan Kaufmann Publishers. xxxii, 1070.
- [3] Hardin, G. (1968). *The Tragedy of the Commons*. Science. **162**: p. 1243–1248.
- [4] Huberman, B.A. and D. Helbing (1999). *Economics-based optimization of unstable flows*. Europhysics Letters. **47**(2): p. 196–202.
- [5] Serman, J.D. (1994). *Learning in and about complex systems*. Cambridge, Mass.: Alfred P. Sloan School of Management, Massachusetts Institute of Technology. 51.
- [6] Chia, M.H., D.E. Neiman, and V.R. Lesser (1998). *Poaching and distraction in asynchronous agent activities*. In the proceedings of Proceedings of the Third International Conference on Multi-Agent Systems. Paris, France. p. 88–95.
- [7] Singh, M.P. (1999). *An Ontology for Commitments in Multiagent Systems: Toward a Unification of Normative Concepts*. Artificial Intelligence and Law. **7**: p. 97–113.
- [8] Jennings, N.R. (1996). *Coordination Techniques for Distributed Artificial Intelligence*, in Foundations of Distributed Artificial Intelligence, G.M.P. O'Hare and N.R. Jennings, Editors. John Wiley & Sons. p. 187–210.
- [9] Gasser, L. (1992). *DAI Approaches to Coordination*, in Distributed Artificial Intelligence: Theory and Praxis, N.M. Avouris and L. Gasser, Editors. Kluwer Academic Publishers. p. 31–51.
- [10] Smith, R.G. and R. Davis (1978). *Distributed Problem Solving: The Contract Net Approach*. Proceedings of the 2nd National Conference of the Canadian Society for Computational Studies of Intelligence.
- [11] Malone, T.W., et al. (1999). *Tools for inventing organizations: Toward a handbook of organizational processes*. Management Science. **45**(3): p. 425–443.
- [12] Klein, M. and C. Dellarocas (2000). *A Knowledge-Based Approach to Handling Exceptions in Workflow Systems*. Journal of Computer-Supported Collaborative Work. Special Issue on Adaptive Workflow Systems. **9**(3/4).
- [13] Klein, M. (2000). *Towards a Systematic Repository of Knowledge About Managing Collaborative Design Conflicts*. In the proceedings of Proceedings of the International Conference on AI in Design (AID-2000). Boston MA. Kluwer Academic Publishers.
- [14] Parsons, S., J.A. Rodriguez-Aguilar, and M. Klein, *A Bluffer's Guide to Auctions*. 2003, MIT Sloan School of Management: Cambridge MA USA.
- [15] Bernstein, A., M. Klein, and T.W. Malone (1999). *The Process Recombinator: A Tool for Generating New Business Process Ideas*. In the proceedings of Proceedings of the International Conference on Information Systems (ICIS-99). Charlotte, North Carolina USA. p. 178–191.

A Framework for Constructing Multi-agent Applications and Training Intelligent Agents

Pericles A. Mitkas, Dionisis Kehagias,
Andreas L. Symeonidis, and Ioannis N. Athanasiadis

Department of Electrical and Computer Engineering,
Aristotle University of Thessaloniki,
GR541 24 Thessaloniki, Greece,
Tel.: +30-2310-996349,
Fax: +30-2310-996398
mitkas@eng.auth.gr,
{diok, asymeon, ionathan}@ee.auth.gr

Abstract. As agent-oriented paradigm is reaching a significant level of acceptance by software developers, there is a lack of integrated high-level abstraction tools for the design and development of agent-based applications. In an effort to mitigate this deficiency, we introduce Agent Academy, an integrated development framework, implemented itself as a multi-agent system, that supports, in a single tool, the design of agent behaviours and reusable agent types, the definition of ontologies, and the instantiation of single agents or multi-agent communities. In addition to these characteristics, our framework goes deeper into agents, by implementing a mechanism for embedding rule-based reasoning into them. We call this procedure “agent training” and it is realized by the application of AI techniques for knowledge discovery on application-specific data, which may be available to the agent developer. In this respect, Agent Academy provides an easy-to-use facility that encourages the substitution of existing, traditionally developed applications by new ones, which follow the agent-orientation paradigm.

1 Introduction

In the last years, agent technology has impressively emerged as a new paradigm for software development [1], which is expected to gain even wider acceptance among the software developers. One important contribution to this effort could be the provision of such tools and development environments that will enable the deployment of agent-based applications quickly and easily. As opposed to this envisioned situation, the current landscape of agent constructing tools is characterized by a plethora of agent development environments, which provide limited capabilities in terms of the level of abstraction in the design and development process of agent-oriented applications. On the other hand, the scope of agent tools and technologies, which dominate the mainstream of development trends in this field, is now becoming clearer than in the past years. In this respect, a

quite desirable effort for agent developers is the creation of a software product that combines all widely used mainstream technologies in one tool. In order to fulfill this demand, we have developed Agent Academy (AA) [2], an integrated framework for constructing multi-agent applications and embedding rule-based reasoning into agents, at the design phase.

The framework presented in this paper is implemented upon the JADE [3] infrastructure, ensuring a relatively high degree of FIPA compatibility, as defined in [4,5]. AA is itself a multi-agent system, whose architecture is based on the GAIA methodology [6]. It provides an integrated GUI-based environment that enables the design of single agents or multi-agent communities, using common drag-and-drop operations. This capability of the AA development environment helps agent application developers to build a whole community of agents with chosen behaviour types and attributes in a few minutes. Using AA, an agent developer can easily go into the details of the designated behaviours of agents and precisely regulate communication properties of agents. These include the type and number of the agent communication language (ACL) messages exchanged between agents, the performatives and structure of messages, with respect to FIPA specifications [7,8,9], as well as the semantics, which can be defined by constructing ontologies with Protégé-2000 [10].

All of the aforementioned characteristics of our development environment have been viewed from an agent-oriented software engineering perspective, since they provide essential elements for the design and the construction of a multi-agent system with pre-specified attributes. In addition to that, there is the AI perspective that deals with the reasoning capabilities of agents. In this context, our system implements a “training module” that embeds essential rule-based reasoning into agents. This kind of reasoning is based on the application of data mining (DM) techniques on possible available datasets. This methodology developed within AA, results in the extraction of agent knowledge in the form of a decision model (e.g. a decision tree). The extracted knowledge is expressed in Predictive Modeling Markup Language (PMML) [11] documents and stored in a data repository, handled by our development framework. The applied data mining techniques are, by definition, updateable as new data come into the repository. Thus, it is easy to update the knowledge bases of agents, by performing agent “retraining”. This capability can be especially exploited in environments with large amounts of periodically produced data. A characteristic example of such an environment is encountered in almost all enterprise IT infrastructures, the vast majority of which are implemented following traditional development paradigms. To this end, our presented infrastructure is envisioned as a convenient tool that will encourage the development of new agent-based applications over the existing traditional ones, by exploiting available data.

The paper is structured as follows. Section 2 briefly reviews related work. Section 3 describes the architecture of our framework and illustrates the development process and the use of tools provided for the construction of a multi-agent system. In section 4, a detailed presentation of the agent “training” mechanism is given. Finally, section 5 concludes the paper and outlines future work.

2 Existing Tools and Applications

The growth in interest and use for agent technology motivated the development of different frameworks and environment to support the implementation of multi-agent systems. Most of them are Java-based applications that aim at facilitating rapid implementation of agent-based applications, by providing mechanisms to manage and monitor message exchanges between agents, and interface support for creating and debugging multi-agent systems.

An advanced open-source tool-kit providing a library of software components and tools that enable the rapid design, development and deployment of agent systems is ZEUS [12]. Although this system is FIPA compliant, it does not support agent mobility, as opposed to AA. Another development environment [13] is implemented as a multi-agent system, in a similar manner as AA, but it does not satisfy the requirements for FIPA compliance.

As far as compliance to FIPA standards is concerned, there is a development framework [14] that meets the FIPA specifications about Agent Management and Agent Communication Language, among others, as well as AA does. Another tool [15] for creating agent systems uses FIPA-ACL for agent messages, but implements its own naming register service, ignoring the relative FIPA specifications.

All of the aforementioned development frameworks do not facilitate the use of any particular reasoning tools, but they do not prevent the agent developers from using other existing tools or implementing their own agent reasoning. In contrast, AA provides both a high-level, GUI-based environment for the design and development of agent-based applications and a training facility that creates rule-based reasoning into the developed agents. A survey of existing tools for creating rule-based reasoning for agents is given in [16].

3 The Development Framework

Our development framework acts as an integrated GUI-based environment that facilitates the design process of a MAS. It also supports the extraction of decision models from data and the insertion of these models into newly created agents. Developing an agent application using AA involves the following activities from the developer's side:

- a. the creation of new agents with limited initial reasoning capabilities;
- b. the addition of these agents into a new MAS;
- c. the determination of existing, or the creation of new behaviour types for each agent;
- d. the importation of ontology-files from Protégé-2000;
- e. the determination of message recipients for each agent.

In case that an agent application developer intends to create a reasoning engine for one or more agents of the designed MAS, two more operations are required for each of those agents:

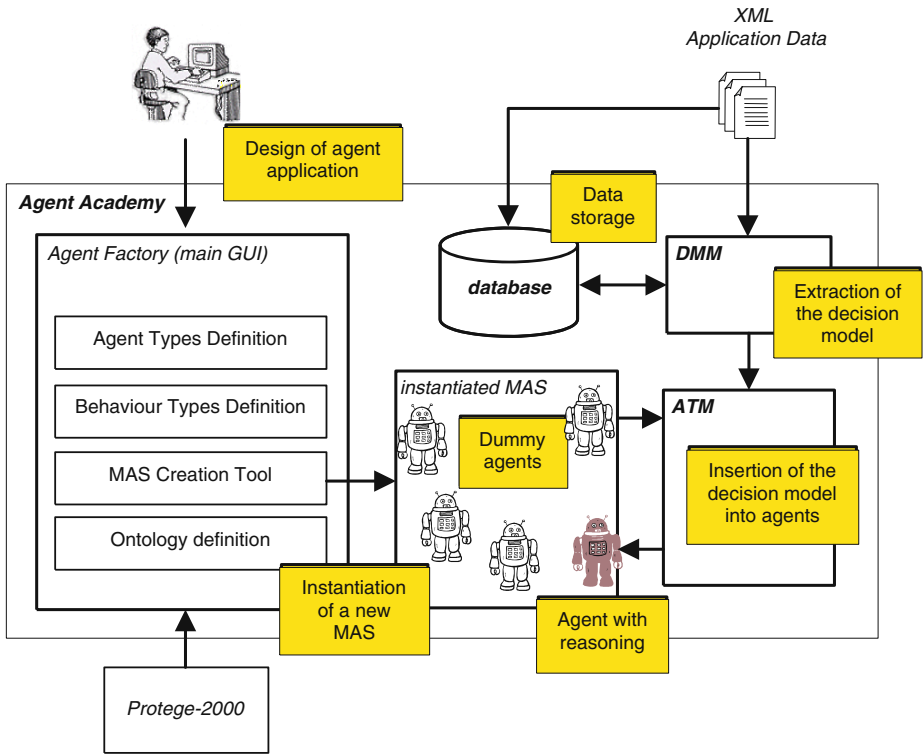


Fig. 1. Diagram of the Agent Academy development framework

- the determination of an available data source of *agent decision attributes*;
- the activation of the training procedure, by specifying the parameters of the training mechanism.

Figure 1 illustrates the Agent Academy main functional diagram, which represents the main components and the interactions between them. In the remaining section, we discuss the Agent Academy architecture, and we explain how the development process is realized through our framework.

3.1 Architecture

The main architecture of AA is also shown in Fig. 1. An application developer launches the AA platform in order to design a multi-agent application. The main GUI of the development environment is provided by the Agent Factory (AF), a specifically designed agent, whose role is to collect all required information from the agent application developer regarding the definition of the types of agents involved in the MAS, the types of behaviours of these agents, as well as the ontology they share with each other. For this purpose, Agent Academy provides

a Protégé-2000 front-end. The initially created agents possess no referencing capabilities (“dummy” agents). The developer may request from the system to create rule-based reasoning for one or more agents of the new MAS. These agents interoperate with the *Agent-Training Module* (ATM), which is responsible for inserting a specific decision model into them. The latter is produced by performing DM on data entered into Agent Academy as XML documents or as datasets stored in a database. This task is performed by the *Data Mining Module* (DMM), another agent of AA, whose task is to read available data and extract decision models, expressed in PMML format.

AA hosts a database system for storing all information about the configuration of the new created agents, their decision models, as well as data entered into the system for DM purposes. The whole AA platform was created as a MAS, which is executed upon JADE.

3.2 Developing Multi-agent Applications

The main GUI of the development platform (Agent Factory) consists of a set of graphical tools, which enable the developer to carry out all required tasks for the design and creation of a MAS, without any effort for writing even a single line of source code. In particular, the Agent Factory comprises the Ontology Design Tool, the Behaviour Type Design Tool, the Agent Type Definition Tool, and the MAS Creation Tool.

3.3 Creating Agent Ontologies

A required process in the creation of a MAS, is the design of one or more ontologies, in order for the agents to interoperate adequately. The Agent Factory provides an *Ontology Design Tool*, which helps developers adopt ontologies defined with the Protégé-2000, a tool for designing ontologies. The RDF files that are created with Protégé are saved in the AA database for further use. Since AA employs JADE for agent development, ontologies need to be converted into special JADE ontology classes. For this purpose, our framework automatically compiles the RDF files into JADE ontology classes.

3.4 Creating Behaviour Types

The *Behaviour Type Design Tool* assists the developer in defining generic behaviour templates. Agent behaviours are modeled as workflows of basic building blocks, such as receiving/sending a message, executing an in-house application, and, if necessary, deriving decisions using inference engines. The data and control dependencies between these blocks are also handled. The behaviours can be modeled as *cyclic* or *one-shot* behaviours of the JADE platform. These behaviour types are generic templates that can be configured to behave in different ways; the structure of the flow is the only process defined, while the configurable parameters of the application inside the behaviour, as well as the contents of the

messages can be specified using the MAS Creation Tool. It should be denoted that the behaviours are specialized according to the application domain.

The building blocks of the workflows, which are represented by nodes, can be of four types:

1. Receive nodes, which enable the agent to filter incoming FIPA-SL0 messages.
2. Send nodes, which enable the agent to compose and send FIPA-SL0 messages.
3. Activity nodes, which enable the developer to add predefined functions to the workflow of the behaviour, in order to permit the construction of multi-agent systems for existing distributed systems.
4. Jess nodes, which enable the agent to execute a particular reasoning engine, in order to deliberate about the way it will behave.

Figure 2 illustrates the design of the behaviour for an agent that receives a message and, according to the content of the message, either executes a pre-specified function, or sends a message to another agent.

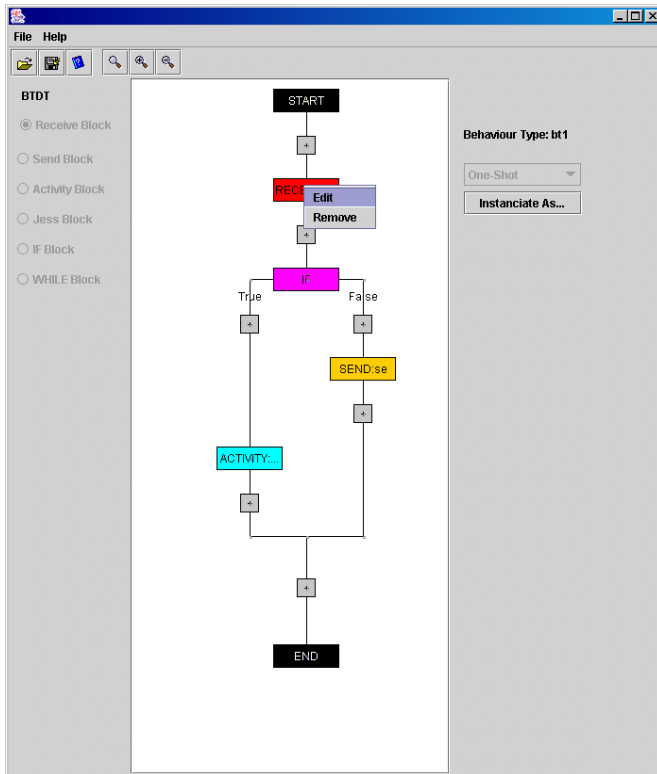


Fig. 2. Creating the Behaviour of an agent through the Behaviour Design Tool

3.5 Creating Agent Types

After having defined certain behaviour types, the *Agent Type Definition Tool* is provided to create new agent types, in order for them to be used later in the MAS Creation Tool. An agent type is in fact an agent plus a set of behaviours assigned to it. New agent types can be constructed from scratch or by modifying existing ones. Agent types can be seen as templates for creating agent instances during the design of a MAS.

During the MAS instantiation phase, which is realized by the use of the MAS Creation Tool, several instances of already designed agent types will be instantiated, with different values for their parameters. Each agent instance of the same agent type can deliver data from different data sources, communicate with different types of agents, and even execute different reasoning engines.

3.6 Deploying a Multi Agent System

The design of the behaviour and agent types is followed by the deployment of the MAS. The *MAS Creation Tool* enables the instantiation of all defined agents running in the system from the designed agent templates. The receivers and senders of the ACL messages are set in the behaviours of each agent. After all the parameters are defined, the agent instances can be initialized. Agent Factory creates *default AA Agents*, which have the ability to communicate with AF and ATM. Then, the AF sends to each agent the necessary ontologies, behaviours, and decision structures.

4 Agent “Training”

The initial effort for the implementation of such a development framework as the one presented in this paper, was motivated by the lack of an agent-oriented software-engineering tool coupled with AI aspects, as far as we know. The ability to incorporate background knowledge into an agent’s decision-making process is arguably essential for effective performance in dynamic environments. However, agent-oriented software engineering methodologies deal with, both high-level, top-down iterative approaches and design methods for software systems [17]. Thus, the lack of tools that concern agent reasoning issues in most high-level software design approaches is excused when we examine these approaches from a pure software-engineering point of view. Moreover, building a MAS with a large number of agents usually requires the reasoning to be distributed in many agents of the MAS community, reducing the degree of reasoning per agent. From our perspective, an agent-oriented development infrastructure should both provide high-level design capabilities and deal with the internals of an agent architecture, in order to be considered complete and generic.

For this reason, we have implemented, as a separate module of the overall agent-oriented development environment a mechanism for embedding rule-based reasoning capabilities into agents. This is realized through the ATM, which is

responsible for embedding specific knowledge into agents. This knowledge is generated as the outcome of the application of DM techniques into available data. The other module, whose role is to exploit possible available datasets in order to extract decision models, is the DMM. Both ATM and DMM are implemented as JADE agents who act in close collaboration.

These two basic modules, as well as the flow of the agent training process are shown in Fig. 3. At first, let us consider an available source of data formatted in XML. The DMM receives data from the XML document and executes certain DM algorithms (suitable for generating a decision model), determined by the agent-application developer. The output of the DM procedure is formatted as a PMML document.

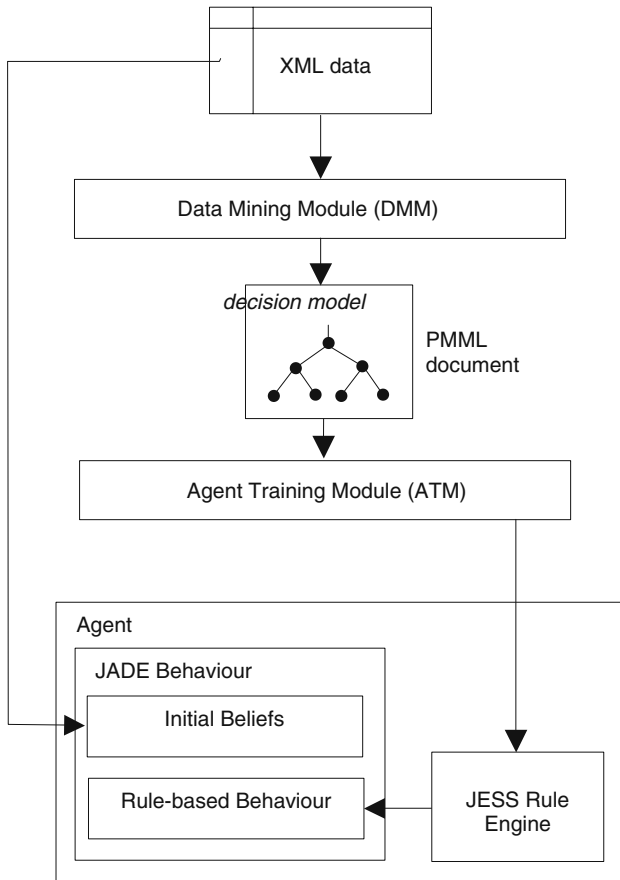


Fig. 3. Diagram of the agent training procedure

PMML is an XML-based language, which provides a rapid and efficient way for companies to define predictive models and share models between compliant vendors' applications. It allows users to develop models within one vendor's application, and use other vendors' applications to visualize, analyze, evaluate or otherwise use the models. The fact that PMML is a data mining standard defined by DMG (Data Mining Group) [11] provides the Agent Academy platform with versatility and compatibility to other major data mining software vendors, such as Oracle, SAS, SPSS and MineIt.

The PMML document represents a knowledge model that expresses the referencing mechanism of the agent we intend to train. The resulted decision model is translated, through the ATM, to a set of facts executed by a rule engine. The implementation of the rule engine is provided by JESS [18], a robust mechanism for executing rule-based reasoning. Finally, the execution of the rule engine becomes part of agent's behaviour.

As shown in Fig. 3, an agent that can be trained through the provided infrastructure encapsulates two types of behaviours. The first is the basic initial behaviour predefined by the AF module. This may include a set of class instances that inherit the *Behaviour* class defined in JADE [5]. The initial behaviour is created at the agent generation phase, using the Behaviour Design Tool, as described in the previous section. This type of behaviour characterizes all agents designed by Agent Academy, even if the developer intends to equip them with rule-based reasoning capabilities. This essential type of behaviour includes the set of initial agent beliefs.

The second supported type of behaviour is the rule-based behaviour, which is optionally created, upon activation of the agent-training feature. This type of behaviour is dynamic and implements the decision model. In the remaining section, we present the details of the data mining procedure and we describe the mechanism for embedding decision-making capabilities into the newly trained agents.

4.1 Mining Background Data for Creating Decision Models

The mechanism for extracting knowledge from available data, in order to provide agents with reasoning, is based on the application of DM techniques on background application-specific data [19]. From our experience with the application of our framework to an industrial scenario about supply chain management [20], we ascertained that the enterprise IT infrastructures generate and manipulate a large amount of data on a permanent basis, thus becoming suitable data providers that satisfy the purposes of DMM.

In the initial phase of the DM procedure, the developer launches the GUI-based wizard depicted in Fig.4(a) and specifies the data source to be loaded and the *agent decision attributes* that will be represented as internal nodes of the extracted decision model. In Fig.4(b) the developer selects the type of the DM technique from a set of available options. In order to clarify the meaning of agent decision attributes, let us consider the decision model in Fig.5. A certain decision is made when some or all input attributes are satisfied. In Fig. 5 we see an input

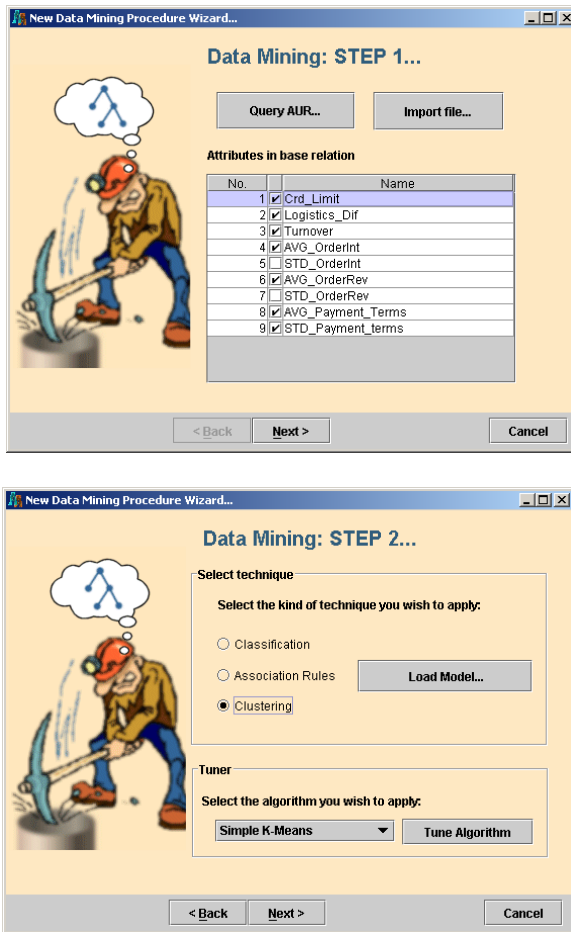


Fig. 4. The two first steps of the DMM wizard

vector of M attributes and an output vector with N attributes, which comprises the overall decision that an agent makes. One part of agent decision attributes is identical to the set of inputs that an agent receives, while the remaining part represents the outputs (decision nodes) of the agent.

Regarding the technical details of the DMM, we have developed the DM facility in our framework, by incorporating a set of DM methods based on the WEKA [21] library and tools and we further extended the WEKA API in order for it to support PMML (a later version of the WEKA API will have our extension included). Some other new DM techniques have also been developed but we will not mention them here, as this would be out of this paper's scope. For further information on the developed DM algorithms, please see [22].

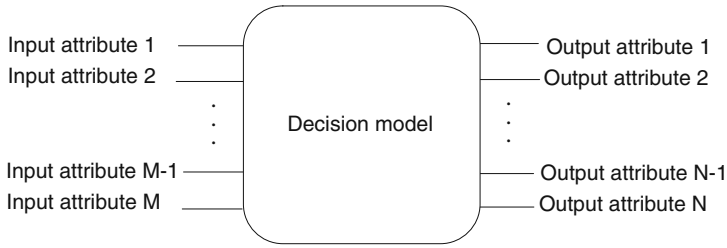


Fig. 5. Input and output attributes in a decision model

In Fig.6(a), we present an XML document, while the respective PMML output, which represents a cluster-based decision model, is shown in Fig.6(b). In order to generate the PMML output, we used the K-means algorithm to perform clustering on selected attributes described in the XML document. The dataset illustrated in Fig.6(a) comes from the design of the formerly mentioned MAS [20] about supply chain management. The XML document concerns customer-related data. The PMML output shown in Fig.6(b), contains, apart from the extracted decision model, some other algorithm-specific details, such as the number of clusters produced, the attributes of the data set and the document version.

4.2 Embedding Intelligence into Agents

We saw in Fig.2 that the completion of the training process requires the translation of the DM resulted decision model into an agent-understandable format. This is performed by the ATM, which receives the PMML output as an ACL message sent by the DMM, as soon as the DM procedure is completed, and activates the rule engine. Actually, the ATM converts the PMML document into JESS rules and communicates, via appropriate messages, with the “trainee” agent, in order to insert the new decision model into it. After the completion of this process, our framework automatically generates Java source code and instantiates the new “trained” agent into the predefined MAS. The total configuration of the new agent is stored in the development framework, enabling future modifications of the training parameters, or even the retraining of the already “trained” agents.

5 Conclusions and Future Work

In this paper we have presented Agent Academy, a multi-agent development framework for constructing multi-agent systems, or single agents. We argued that the existing tools and infrastructures for agent development are especially focused on the provision of high-level design methodologies, leaving out the details of agents’ decision-making abilities. In contrast, our framework can provide both a GUI-based, high-level MAS authoring tool and a facility for extracting rule-based reasoning from available data and inserting it into agents. The

```

<Instances title="ALTEC Data" author="Kehagias Dionisis">
  <Attributes>
    <Attribute>
      <Name>AVG_Order_Freq</Name>
      <Type>numeric</Type>
    </Attribute>
    <Attribute>
      <Name>AVG_Order_Rev</Name>
      <Type>numeric</Type>
    </Attribute>
    <Attribute>
      <Name>AVG_Payment_Trms</Name>
      <Type>numeric</Type>
    </Attribute>
    <Attribute>
      <Name>Crd_Limit</Name>
      <Type>numeric</Type>
    </Attribute>
    <Attribute>
      <Name>Logistics_Dif</Name>
      <Type>numeric</Type>
    </Attribute>
    <Attribute>
      <Name>STD_Order_Freq</Name>
      <Type>numeric</Type>
    </Attribute>
    <Attribute>
      <Name>STD_Order_Rev</Name>
      <Type>numeric</Type>
    </Attribute>
    <Attribute>
      <Name>STD_Payment_Terms</Name>
      <Type>numeric</Type>
    </Attribute>
    <Attribute>
      <Name>Turnover</Name>
      <Type>numeric</Type>
    </Attribute>
  </Attributes>
  <Data>
    <Row>
      <Crd_Limit>-8.23599E-2</Crd_Limit>
      <Logistics_Dif>-6.69967E-2</Logistics_Dif>
      <Turnover>-0.138325</Turnover>
      <AVG_Order_Freq>-0.64769</AVG_Order_Freq>
      <STD_Order_Freq>-0.71325</STD_Order_Freq>
      <AVG_Order_Rev>-0.35288</AVG_Order_Rev>
      <STD_Order_Rev>-0.21821</STD_Order_Rev>
      <AVG_Payment_Trms>-1.32909</AVG_Payment_Trms>
      <STD_Payment_Terms>
        0.50519</STD_Payment_Terms>
      </Row>
    </Data>
  </Instances>

```

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE pmml_2_0.dtd>
<PMML>
  <Header copyright="CERTH" description=" Clustering
  Model of ERP data">
    <Application name="Agent Academy" version="0.3" />
  </Header>
  <DataDictionary numberOfFields="9">
    ...
  </DataDictionary>
  <ClusteringModel
  modelName="ERP-org.agentacademy.modules.dataminer.f
  ilters.ReplaceMissingValuesFilter"
  modelClass="centerBased" numberOfClusters="5">
    <MiningSchema>
      ...
    </MiningSchema>
    <ClusteringField field="AB"
    compareFunction="squaredEuclidean" />
    <ClusteringField field="CL"
    compareFunction="squaredEuclidean" />
    <ClusteringField field="TO"
    compareFunction="squaredEuclidean" />
    <ClusteringField field="AOP"
    compareFunction="squaredEuclidean" />
    <ClusteringField field="STDAOP"
    compareFunction="squaredEuclidean" />
    <ClusteringField field="AOI"
    compareFunction="squaredEuclidean" />
    <ClusteringField field="STDAOI"
    compareFunction="squaredEuclidean" />
    <ClusteringField field="APT"
    compareFunction="squaredEuclidean" />
    <ClusteringField field="STDAPT"
    compareFunction="squaredEuclidean" />
    <Cluster name="0">
      <Array n="9"> -1.825885 17.36695 18.23353 -0.55470
      -0.47161 7.96735 14.47850 0.40154 0.92600</Array>
    </Cluster>
    <Cluster name="1">
      <Array n="9">
        0.06525 -0.27008 -0.12450 -0.64679 -0.71163 -0.35276 -0.218
        16 -1.32502 -0.49305</Array>
    </Cluster>
    <Cluster name="2">
      <Array n="9"> 0.03786 0.04243 0.04831 -0.08052
      -0.17864 0.16450 0.09465 0.63661 0.00631 </Array>
    </Cluster>
    ...
  </ClusteringModel>
</PMML>

```

Fig. 6. XML input (a) and PMML output (b)

produced knowledge is expressed as PMML formatted documents. We have presented the functional architecture of our framework; we shortly demonstrated an indicative scenario for deploying a MAS and, finally, we discussed the details of the agent “training” process.

Through our experience with Agent Academy, we are convinced that this development environment significantly reduces the programming effort for building

agent applications, both in terms of time and code efficiency, especially for those MAS developers who use JADE. For instance, one MAS, that requires the writing of almost 6,000 lines of Java code, using JADE, requires less than one hour to be developed with Agent Academy. This test indicates that AA meets the requirement for making agent programs in a quicker and easier manner. On the other hand, our experiments with the DMM have shown that the completion of the decision model generated for agent reasoning is highly dependant on the amount of available data. In particular, a dataset of more than 10,000 records is adequate enough for producing high-confidence DM results, while datasets with fewer than 3,000 records have yielded non-consistent arbitrary output.

The AA framework is the result of a development effort, which begun two years ago. Currently, a beta version exists, which is not yet publicly available. The first stable implementation of AA is planned to come out on July 2003, as an open-source product. Our near future work involves the finalization of the integration process for AA, as well as the exhaustive testing of the platform, by implementing three large-scale applications in the domains of real-time notification, web-based applications, and supply-chain management, respectively.

Acknowledgements. Work presented in this paper is partially funded by the European Commission, under the IST initiative as a research and development project (contract number IST-2000-31050, “Agent Academy: A Data Mining Framework for Training Intelligent Agents”). Authors would like to thank all members of the Agent Academy consortium for their remarkable efforts in the development of such a large project.

References

1. Lind, J.: Issues in agent-oriented software engineering. In: First International Workshop on Agent-Oriented Software Engineering (AOSE-2000), Limerick, Ireland (2000)
2. Agent Academy Consortium: Agent Academy. (2000) Available at: <http://agentacademy.iti.gr>.
3. Bellifemine, F., Poggi, A., Rimassa, G., Turci, P.: An object-oriented framework to realize agent systems. In: Proceedings of WOA 2000 Workshop, Parma, Italy (2000) 52–57
4. Foundation for Intelligent Physical Agents: FIPA Developer’s Guide. (2001) Available at: <http://www.fipa.org/specs/fipa00021/>.
5. Bellifemine, F., Caire, G., Trucco, T., Rimassa, G.: JADE Programmer’s Guide. (2001) Available at: <http://sharon.csel.it/>.
6. Wooldridge, M.J., Jennings, N.R., Kinny, D.: The gaia methodology for agent-oriented analysis and design. *Journal of Autonomous Agents and Multi-Agent Systems* **3** (2000) 285–312
7. Foundation for Intelligent Physical Agents: FIPA Communicative Act Library Specification. (2001) Available at: <http://www.fipa.org/specs/fipa00037/>.
8. Foundation for Intelligent Physical Agents: FIPA SL Content Language Specification. (2002) Available at: <http://www.fipa.org/specs/fipa00008/>.

9. Foundation for Intelligent Physical Agents: FIPA ACL Message Structure Specification. (2002) Available at: <http://www.fipa.org/specs/fipa000037/>.
10. Noy, N.F., Sintek, M., S., D., Crubezy, M., Ferguson, R.W., Musen, M.A.: Creating semantic web contents with Protégé-2000. *IEEE Intelligent Systems* **16** (2001) 60–71
11. Data Mining Group: Predictive Model Markup Language Specifications (PMML), ver. 2.0. (2002) Available at: <http://www.dmg.org>.
12. Nwana, H., Ndumu, D., Lee, L., Collis, J.: ZEUS: A tool-kit for building distributed multi-agent systems. *Applied Artificial Intelligence Journal* **13** (1999) 129–186
13. Gutknecht, O., Ferber, J.: Madkit: A generic multi-agent platform. In: 4th International Conference on Autonomous Agents, Barcelona, Spain (2000)
14. Suguri, H., Kodama, E., Miyazaki, M., Nunokawa, H., Noguchi, S.: Madkit: A generic multi-agent platform. In: Proceedings of the Workshop on Ontologies in Agent Systems, 5th International Conference on Autonomous Agents, Montreal, Canada (2001)
15. Jeon, H., Petrie, C., Cutkosky, M.: JATLite: a java agent infrastructure with message routing. *IEEE Internet Computing* **4** (2000) 87–96
16. Rahimi, S., Cobb, M., Ali, D., Paprzycki, M.: An analysis of intelligence-enhancing techniques for software agents. In: Proceedings of the 5th World Multi-Conference on Systemics, Cybernetics and Informatics, Orlando (2001)
17. Tveit, A.: A survey of agent-oriented software engineering. In: Proceedings of the NTNU Computer Science Graduate Student Conference, Norwegian University of Science and Technology, Trondheim, Norway (2001)
18. Friedman-Hill, E.: Java Expert System Shell (JESS). Sandia National Laboratories. (2002) Available at: <http://herzberg.ca.sandia.gov/jess>.
19. Symeonidis, A.L., Mitkas, P.A., Kehagias, D.: Mining patterns and rules for improving agent intelligence through an integrated multi-agent platform. In: 6th IASTED International Conference, Artificial Intelligence and Soft Computing ASC, Banff, Alberta, Canada (2002)
20. Symeonidis, A.L., Kehagias, D., Koumpis, A., Vontas, A.: Open source supply chains. In: 10th International Conference on Concurrent Engineering (CE-2003), Workshop on intelligent agents and data mining: research and applications, Madeira, Portugal (2003)
21. Witten, I.H., Frank, E.: Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations. Morgan Kaufmann Publishers, San Francisco, CA (2000)
22. Athanasiadis, I.N., Kaburlasos, V.G., Mitkas, P.A., Petridis, V.: Applying machine learning techniques on air quality data for real-time decision support. In: First International NAISO Symposium on Information Technologies in Environmental Engineering (ITEE'2003), Gdansk, Poland (2003)

Activity Theory for the Analysis and Design of Multi-agent Systems

Rubén Fuentes¹, Jorge J. Gómez-Sanz¹, and Juan Pavón¹

Universidad Complutense Madrid, Dep. Sistemas Informáticos y Programación
28040 Madrid, Spain
{ruben, jjgomez, jpavon}@sip.ucm.es
<http://grasia.fdi.ucm.es>

Abstract. Modeling a Multi-Agent System (MAS) involves a large number of entities and relationships. This implies the need for defining an organization, in order to structure and manage the complexity of the whole system. In this work we propose the use of human organization metaphors and their application to the verification of MAS specification. In concrete, we apply Activity Theory, which has its roots in Sociology, to study agent systems and obtain relationship patterns that can be applied to the analysis of MAS. These patterns guide analysis and design refinements, and help to detect inconsistencies. This technique has been implemented and integrated in the INGENIAS IDE platform, and proved with some case studies, in particular, for agent-based web applications.

1 Introduction

The agent-oriented paradigm provides a new perspective for the development of complex software systems. However, despite the advantages of using the agent abstraction instead of traditional system/subsystem decomposition, building Multi-Agent Systems (MAS) is not trivial at all. Describing a complex MAS will result in a considerable number of interrelated diagrams (like in INGENIAS [15] or MAS-CommonKADS [8]), or a few large diagrams (like in Tropos [2] or GAIA [18]).

As the number of diagrams and elements that participate in the specification of a MAS grows, the probability that a contradiction appears increases also. Here *contradiction* stands for an inconsistency, obscure point, or misunderstanding in the models that describe the MAS. To identify these contradictions, a developer may consider the use of formal methods. However, the application of formal methods requires skills that are not frequent among developers. As a solution, this paper proposes another abstraction that allows simple reasoning about a specification.

This approach starts with the assumption that agents have a strong social component [12, 16] and that managing the complexity of a MAS requires structuring the system as an organization [5]. Following this line, and considering that agents can be used to simulate human organizations, we think that theories that explain human

¹ This work has been developed in the project INGENIAS (TIC2002-04516-C03-03), which is funded by Spanish Ministry of Science and Technology.

organizations may also be applied to MAS. In this sense, we have studied Activity Theory (AT) [11] as a technique for analysing MAS. The use of this social theory is feasible on agent-oriented methodologies as they have an intentional and collective component similar to what AT assumes on human organizations. AT identifies regular patterns, some of which are contradictions, that appear in human organizations, and with them it explains how social systems change or why sometimes they do not behave as expected. This way, from the AT perspective, contradictions guide system development, since they identify both problems and solutions. The use of AT also reduces the communicative gap between developers and customers. Analysis abstractions, even the agent one, are difficult to understand by non-experts. On the contrary, AT social abstractions are closer to both customers and developers as they represent common knowledge about human communities. For this reason, AT concepts build a useful communicative language for the development process.

AT, as it has been developed in Sociology, works with specifications in unstructured natural language. To allow a more systematic use, we have adapted its concepts to a well-known formalism in software engineering: UML [14]. This kind of notation provides an intuitive and expressive tool for communication of AT concepts. Using UML diagrams, we formalize AT contradictions as structural patterns. Developers study these patterns looking for correspondences in their models, which could be described in any language. If correspondences appear, they give a graphical structural explanation of the contradiction in terms of model elements. This explanation helps to understand the contradiction and find a solution. However, AT contradiction patterns do not have to be expressed with the same specification language as the one used, for instance, for the analysis specification. For this reason, the application of the proposed approach requires first to define mappings from AT concepts to those used in the agent modeling technique of choice.

The following section justifies the use of the contradictions as a driver for system development. Then, section 3 explains why AT can be used to find contradictions in MAS. Section 4 shows how contradictions can be represented by structural patterns, and section 5 provides some examples of it with AT. In section 6, previous elements are integrated in a validation method that checks the consistency of the different elements and parts of a MAS specification against a set of patterns. This is illustrated with an example in the following section. Finally, conclusions discuss the possibility of applying AT for MAS checking and the value of analyzing contradictions taking into account the results of the experimentation.

2 Contradictions as a Motor of Change

Contradictions are inherent to the software development process. They appear no matter how disciplined a developer is. Contradictions arise in several ways that require different treatments. According to their origin, contradictions can be classified as:

- **External.** There are contradictions which have their origin in the environment. Different systems running in the same environment may influence negatively among themselves.

- **Internal.** This situation may occur in a MAS when there are conflicting system goals or different agents in the system pursue incompatible goals.
- **Analysis mistakes.** A software system is the result of a collaboration among customers and developers. These groups of people usually have different backgrounds and this causes misunderstandings in their communication. Besides, real complex systems are described, at least in conventional software engineering, from several viewpoints. It is difficult to maintain all of them updated and consistent.

Contradictions are not just problems to solve. According to Activity Theory [11], they can be the principle for guiding a development, the form in which the development progresses. This means that new versions of the system specification emerge as solutions to the contradictions of a previous one.

3 Activity Theory Applied to Intentional Systems

As systems that work in the knowledge level, MAS are intentional and social [12, 16]. By intentional, we mean that the system follows the Rationality Principle [13]: Every action taken by an agent aims at achieving one of its goals. According to sociology, these features also appear in human organizations. As a consequence of this common base of both human organizations and MAS, social theories can be examined as a source of information to help finding contradictions in MAS. One of them is Activity Theory [11].

Activity Theory (AT) is a framework for the study of different forms of human practices, as developmental processes with both the individual and social levels interleaved. From an AT point of view, people are embedded in a socio-cultural context and their behaviour cannot be understood independently of it. But people do not have just a passive relationship with the surrounding context; they actively interact with it, changing the objects and creating new artefacts. This complex interaction of individuals with their surroundings has been called *activity* and is considered as the fundamental unit of analysis.

To better understand the concepts behind AT that apply to MAS, we have represented the core of AT using UML (see Fig. 1). The graphical notation employed with Activity Theory uses a simple colour code to distinguish the different stereotypes that a component of a specification can use.

The *activity* [4] is the central analysis concept and it reflects a process. The *subject* is the active element that carries out the *activity*. It can represent an individual or a group of individuals. The *subject* has some definite needs represented by the *objective*. The *objective* is satisfied by the *outcome*, which is produced transforming an *object*. The *object* may be an ideal or material one. So, the *activity* is a process executed by the *subject*, to transform an *object* in the required *outcome* that satisfies *subject's objective*. In order to carry out that transformation, the *subject* employs *tools*: *tools* always mediate *subject's* processes over *objects* [17]. A *tool* can be anything used in the process, including both material tools (e.g., a computer) and tools for thinking (e.g., a plan). The *tool* is at the same time enabling and limiting in the *activity*: it empowers the *subject* in the transformation process with the historically collected experience and skills crystallised to it; but it also restricts the interaction to

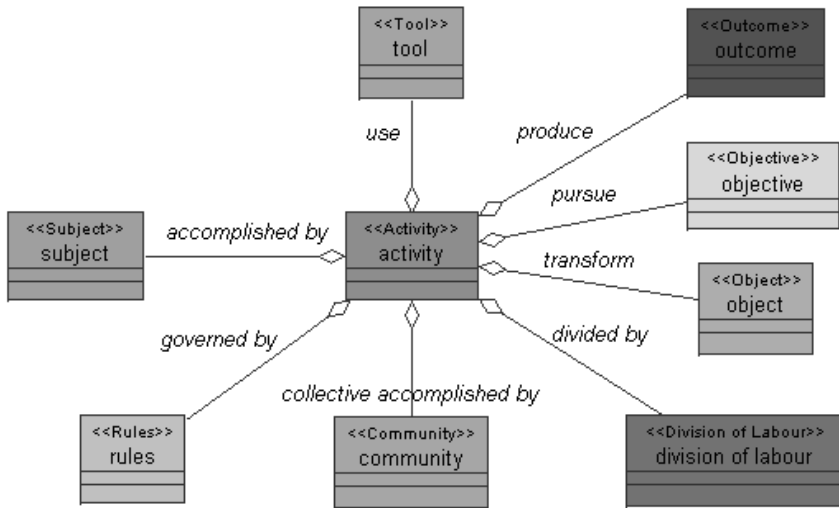


Fig. 1. Activity theory concepts and mediated relationships

be from the perspective of that particular *tool*, that is, the instrument hides other potential features of an *object* to the *subject*.

The social level is characterised by the three lower concepts in the diagram (Fig. 1). The *community* represents those *subjects* who share the same *object* [10]. *Rules* mediate between *subject* and *community*, and the *division of labour* between *object* and *community*. An *activity* may concern many *subjects* and each *subject* may play one or more roles and have multiple motives. *Rules* specify how *subjects* fit into *communities*. *Rules* cover both explicit and implicit norms, conventions, and social relations within a *community*. *Division of labour* refers to the explicit and implicit organisation of the *community* as related to the transformation process of the *object* into the *outcome*.

As a generalization, an *artefact* states for whatever kind of concept in AT.

Using the previous notation with some additional concepts, we express what kinds of contradictions and solutions AT identifies. The additional elements represent contribution relationships and are related with i^* [19]. Contributions show how different *artefacts* have influence over others. Examples of this kind of relationships are satisfaction of *objectives*, construction of *objects*, or damage of *tools*. Added relationships that represent them are: *contribute* (*positively*, *negatively* or *undefined*), *essential* and, *impede*.

4 Contradictions as Patterns

When a developer checks a software model, he intends to detect contradictions, give them an explanation, and, possibly, investigate how the checked model should be changed. To facilitate this work it is important to choose the appropriate language.

Many modeling techniques are supported by visual languages. These visual languages can be regarded as semi-formal since they are not completely unambiguous. They often offer an accurate tradeoff between their communicative and expressive power, which makes them attractive for development. Their representations are intuitive enough to provide the tools for communication inside the development team and with customers. At the same time, they have a precise enough semantics to support basic validation. Of course, further validation capabilities are desirable provided that usability is not compromised.

In order to gain these additional capabilities and keep usability, we propose the description of contradictions as patterns using visual languages. The meaning of a contradiction pattern comes from an interpretation of the primitives of the visual language. To be able to use these patterns in a model, belonging to an analysis or a design, and find contradictions, we need a mapping between vocabularies in the model and the elements in the contradiction pattern. With contradiction patterns, mappings, and a model to validate, detection of inconsistencies consists on finding the part of the model that matches the structure of the contradiction pattern.

There are three main advantages of managing contradictions as patterns:

1. Patterns identify the elements and structures to find in the model in order to detect contradictions. The combined use of patterns and mappings simplifies the detection process to a correspondence one.
2. The same approach provides hints to the solution for contradictions. Pattern definition comes together with one or several solution patterns. These solutions are inspired by the experience of the same social theories that provide contradictions. These theories have studied such problems for a long time. Working in the same way with contradictions and their solutions gives a uniform path through the whole validation process, from detection, to explanation and solving.
3. A graphical notation describes inconsistencies. This kind of notations helps in the communication between customers and developers. There are many successful examples of its use, for instance, UML [14], Petri Nets [3], or Tropos [2].

Contradiction patterns, as a validation tool, are not intended to show how a MAS should be build, like design patterns in [6], but to point out conflictive configurations in the analysis.

5 AT Patterns

Using the concepts and notation presented in the previous section, AT contradictions are described as graphical patterns. AT research [4,10,11,17] describes several conflictive situations in human *activities*. Two of the most referenced contradictions are *Double bind* and *Twofold meaning*.

Double bind (see Fig. 2) represents a situation in which no matter what the *subject* does, every action has some negative effect on at least one of its *objectives*. The *subject* is able to carry out several *activities* but each one has some related goal that is negatively affected by that action.

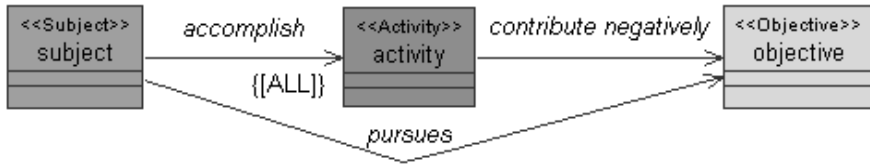


Fig. 2. Double bind contradiction

One possible solution to a *Double bind* situation is to decompose *activities* with additional ones to minimize those negative contributions to *objectives*. An *activity* can be decomposed in different sequences of *activities*. Although some of the *activities* in the sequence will probably preserve the negative effect over *objectives*, it is possible to introduce others with a positive effect. The model could be changed to include this information, as it is shown in Fig. 3.

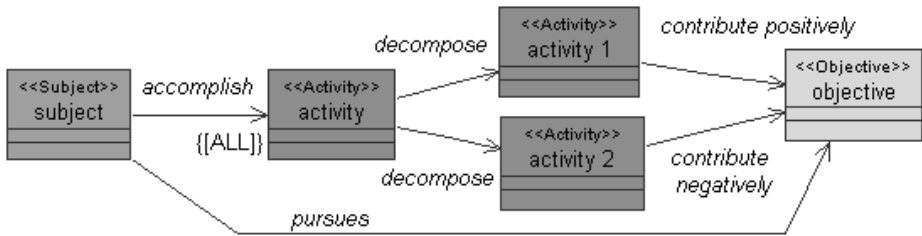


Fig. 3. One possible solution to *Double bind* contradiction

A *twofold meaning* situation (see Fig. 4) emerges when an artefact has several different uses inside an activity. If these uses are not consistent, an inner contradiction arises. This contradiction does not imply always an analysis mistake. It can draw attention to the need of discussing some aspects of the system with customers, or detail possible expansion points in the analysis.

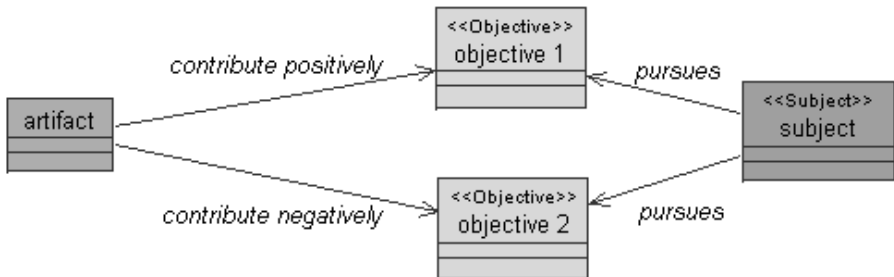


Fig. 4. *Twofold meaning* contradiction

One possible solution to a *Twofold meaning* contradiction is to refine *artifact* contributions to *objectives*. If the *artifact* has to be preserved in the system is because it is essential to one of the *objectives*. If that *objective* does not exist, the *artifact* should be removed. At the end, the model could appear as seen in Fig. 5.

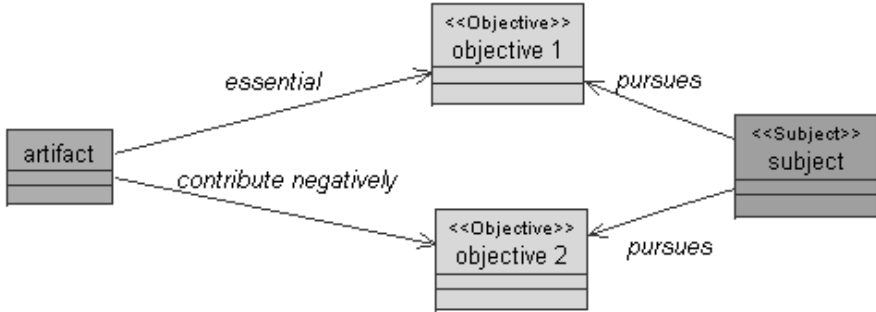


Fig. 5. One possible solution to *Twofold meaning* contradiction. Note the different labels in the edges from *artifact* to *objectives* with respect to Fig. 4.

6 Model Validation Method

This section presents a general method to check a model against a set of patterns. This process will be used to apply the ideas of previous sections to a real specification. The specification uses INGENIAS [15] as notation, which is an evolution of MESSAGE [1]. We have chosen INGENIAS for several reasons. One is to show that it is possible to use a different notation to express the AT patterns (i.e., the one presented in this paper) and to model the MAS (i.e., INGENIAS in this case). Another is the availability of the INGENIAS IDE [9], which allows us to implement and integrate the validation of AT patterns with existing agent development tools.

The proposed process has two parameters: the contradictions to check and a set of views that describe a MAS.

The first step of the process is to establish a mapping between the vocabulary used in the analysis and the one of AT. Since AT and MAS methodologies model intentional and social systems, the mapping is straightforward. Fig. 6 shows simple mappings from AT to INGENIAS abstractions. Note that some artefacts from AT correspond to several artefacts in INGENIAS. This happens because AT concepts are more general than those in INGENIAS.

With these mappings, models are traversed looking for groups of entities and relationships that fit into contradiction patterns. When one of these groups is found, the pattern provides further information of what is the problem and what solutions exist.

Activity Theory	INGENIAS
Activity	Task, Workflow
Subject	Agent, Group, Organization
Object	Resource, Application, Fact
Outcome	Resource, Application, Fact
Objective	Goal
Tool	Resource, Application
Community	Group, Organization
Rules	Mental states
Division of Labour	Mental states

Fig. 6. Mappings between Activity Theory and INGENIAS

The third step is to solve the problem, if possible. Patterns not only point out inconsistencies but also related patterns that explain how to solve them. For example, a *twofold meaning* contradiction can be solved showing that an artefact is essential to carry out a task despite of its negative effects, or through the introduction of further refinement in the goals set.

7 Testing the Patterns

The case study used here as example to show the application of AT patterns in MAS analysis, is a recommender system that relies on collaborative filtering techniques, which is described in [7]. The full specification of the whole system can be found at <http://grasia.fdi.ucm.es/ingenias>.

This collaborative filtering system assumes that if a user finds interesting a piece of information, then other users with similar opinion and preferences may also find interesting that piece of information.

In this system, there are two kinds of agents:

- Community agents. They represent a community of users that share common interests. They are responsible of users' management and information broadcast.
- Personal agents. They represent individuals who belong to the community. A Personal Agent can be subscribed to several communities. This kind of agents is the interface between users and the community in search and collaboration processes.

A Personal Agent plays several roles in the system. Two of these roles (see Fig. 7) are related with requests of subscription to communities and suggestion of information.

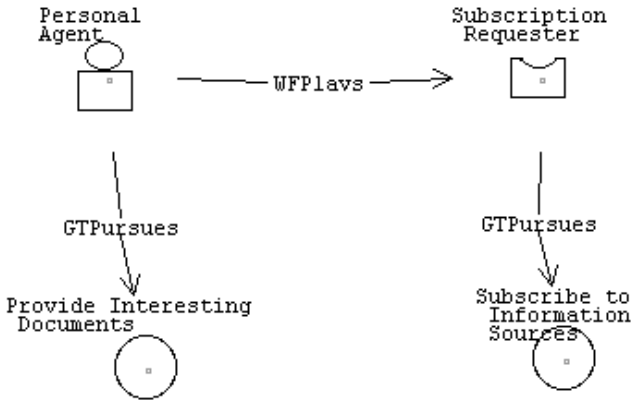


Fig. 7. A *Personal Agent* playing the role *Subscription Requester*, and their goals. The icon in the upper left corner represents an agent, the one in upper right corner denotes a role, and circles symbolize goals. This figure has been created with the INGENIAS IDE [9]

A *Personal Agent* has the goal of *Provide Interesting Documents*. This goal is necessary in order to supply information to a community of users. Besides, it tries to find new information sources and subscribe to them. This is represented with the role *Subscription Requester* that pursues the goal of *Subscribe to Information Sources*. In this case, the goals have no mutual dependencies in the original model except belonging to the decomposition of a very general system goal. Workflows that satisfy these goals have neither outputs nor tasks in common (see Fig. 8). These workflows and their connection are shown as a producer-consumer chain of tasks and agent mental entities (what is produced and consumed).

The connection of both workflows comes through the use of an external application, the *Statistics Manager*, and the mental entity *Subscribed Communities*. *Subscribed Communities* represents the information sources of an agent. When the *Personal Agent* plays the role *Subscription Requester*, it pursues the goal *Subscribe to Information Sources*. This goal intends to increase information sources, and therefore, the number of communities registered in *Subscribed Communities*. Nevertheless, the *Personal Agent* also pursues *Provide Interesting Documents* to the community. To satisfy this goal, the agent executes a workflow in which their suggestions are registered with the application *Statistics Manager*. These statistics are then used to evaluate the *Personal Agent*. If this evaluation is negative, the agent is expelled from the community, what decreases the number of communities in *Subscribed Communities*. So, make suggestions could have a negative side effect over obtaining information sources. The agent is pursuing two goals which affect the same entity in an opposite way.

The previous contradiction can be described in terms of AT *twofold meaning* contradiction (see Fig. 9). The activity that satisfies *Provide Interesting Documents* has influence over goal *Subscribe to Information Sources*. It modifies the entity *Subscribed Communities*, which is related with *Subscribe to Information Sources*.



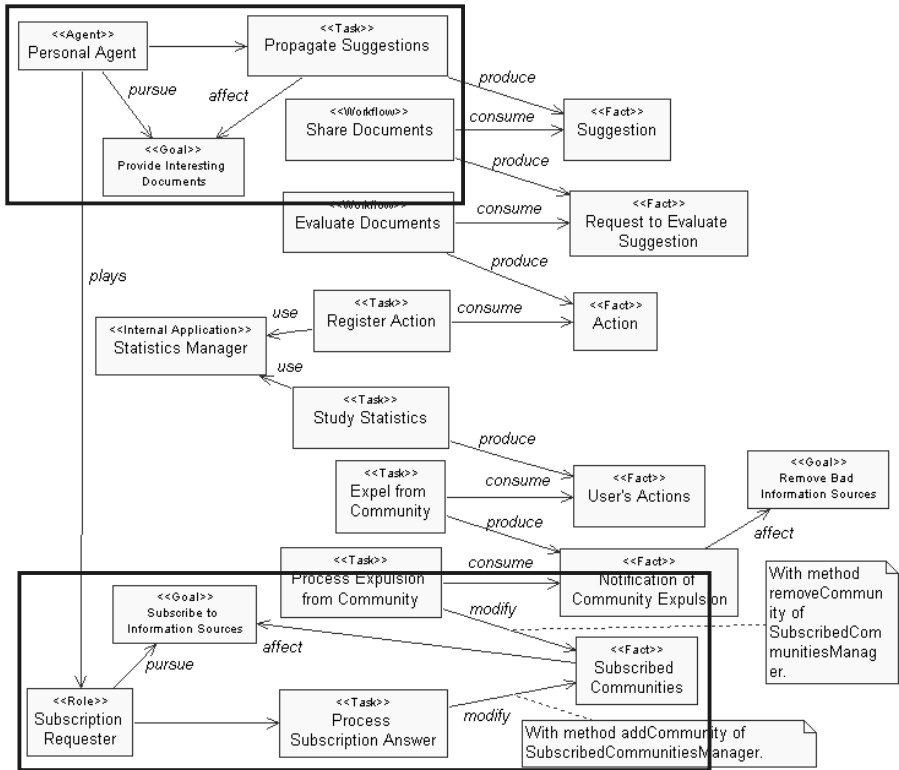


Fig. 8. Workflows related with *Personal Agent* goals of *Provide Interesting Documents* and *Subscribe to Information Sources*. Stereotypes show the type of entities according to INGENIAS notation. Conflicting regions are enclosed by a blue rectangle.

Besides, these goals are not related by contribution relationships. Without any information about the effect of involved tasks, it is only possible to conclude that this workflow could be carrying out contradictory tasks and producing side effects over *Subscribe to Information Sources*.

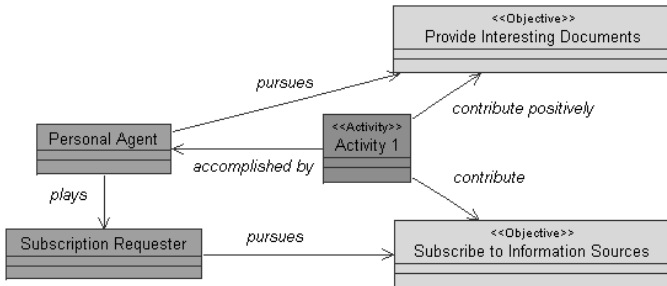


Fig. 9. Twofold meaning contradiction for *Personal Agent*

At this point, existing diagrams (see Fig. 7, Fig. 8, and Fig. 9) do not provide information about the semantics of operations over *Subscribed Communities*. So the customer should be asked about relationships between goals, the entity, and the activity. The customer could add additional information over the contradiction diagram (see Fig. 10) to clarify the analysis.

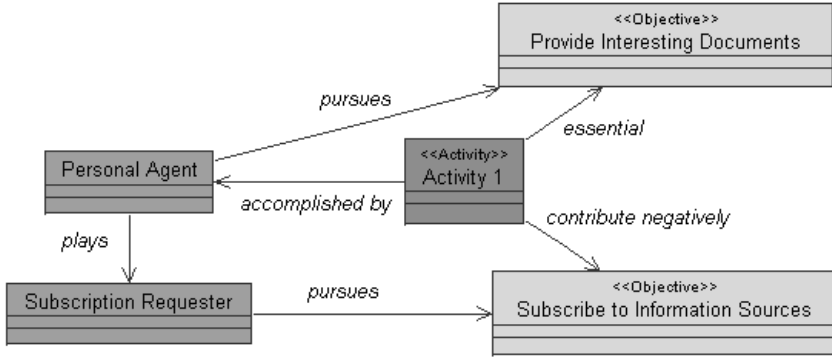


Fig. 10. Twofold meaning contradiction for *Personal Agent* with additional information

The graphical model shows that the contradiction is inherent to the system. The *Personal Agent* has to carry out the activity to satisfy its goal *Provide Interesting Documents*. The customer could argue that the negative side effect is due to the need to evaluate user’s suggestions. This evaluation could cause the expelling of the agent from the community but it is needed to preserve the quality of global information.

In this case, the negative effect may not have a solution for the *Personal Agent*. Despite of this, the use of the patterns has allowed a deeper comprehension of the contradiction which could cause a rethinking of the involved workflows. To make it, this process should be combined with conventional analysis techniques, as it is a tool to help in a MAS development process.

8 Conclusions

This paper has shown an approach for MAS verification that is based on the application of Activity Theory. The verification process consists on checking a MAS specification against a set of contradiction patterns. Such approach starts from the hypothesis that misunderstandings, inconsistencies, and mistakes are inherent to the development of complex software systems. Instead of looking at them just as errors, contradictions can be considered as another steps in the development process. Also, detecting contradictions allows establishing what parts of an analysis need further refinements or corrections.

The use of AT has helped to understand and explain contradictions in a MAS development process. The main characteristics of this approach are:



- The use of AT gives a real source of contradictions having their roots in the social component of MAS. This paper presents two of them, *Double bind* and *Twofold meaning*. Moreover, the proximity of social concepts and humans helps both customers and developers to analyze MAS models.
- Working with contradictions as structural patterns to match against models has several advantages. These patterns allow a reuse of the knowledge they contain over different methodologies. The only requirement to apply this method is that the methodology has to be based in the agent paradigm. This makes possible to build the needed mapping between modeling concepts and the contradiction patterns language. Besides this, a contradiction pattern points out what to look for, and, when found, its correspondence with the model explains the problem. Finally, the structural approach gives also a simple way to express possible solutions to the problem according to the context.
- A graphical notation allows to bridge the gap between experts and non-experts. Here we use a widespread language, UML, as the basis to express these contradiction patterns.

The experimentation has exposed three main concerns when applying this approach, which are the subject for future work. The first one is how to detect if a contradiction is meaningful. In the case of the *Twofold Meaning*, the analyst needs customer's assistance to decide if the contradiction exists or not. The customer is the person who really knows the problem domain and the meaning of concepts in the real world. But it is a very hard work for a non-expert to understand a software analysis: the customer needs the analyst's help. Here emerges the second issue: tools used to express contradiction patterns have to be understandable and accurate both for customers and analysts. The third issue is related with the development process, how to make it incremental. If contradictions are going to guide the whole process, they should be applicable from requirement elicitation to design with different levels of detail.

References

1. Caire, G., Coulier, W., Garijo, F. Gomez, J., Pavon, J., Leal, F., Chainho, P., Kearney, P., Stark, J., Evans, R., and Massonet, P.: Agent Oriented Analysis using MESSAGE/UML. In: Wooldridge, M., Weiss, G., and Ciancarini, P. (Eds.): The Second International Workshop on Agent-Oriented Software Engineering (AOSE 2001). Lecture Notes in Computer Science, Vol. 2222. Springer-Verlag (2002) 119–135.
2. Castro, J., Kolp, M., and Mylopoulos, J.: *A Requirements-Driven Development Methodology*. In: Proc of the 13th International Conference on Advanced Information Systems Engineering CAISE'01, Interlaken, Switzerland, June 4-8, 2001.
3. Cost, R. S., Chen, Y., Finin, T., Labrou, Y., and Peng, Y.: *Using Colored Petri Nets for Conversation Modeling*. In: Issues in Agent Communication. Springer Verlag, 2000.
4. Engeström, Y.: *Learning by expanding*. Orientakonsultit, Helsinki. 1987.
5. Ferber, J., Gutknecht, O., and Michel, F.: From Agents to Organizations: an Organizational View of Multi-Agent Systems. In this book.
6. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley Professional Computing Series. 1995.

7. Gómez-Sanz, J., Pavón J. and Díaz-Carrasco, A.: The PSI3 Agent Recommender System. In: Cueva J.M. et al. (Eds.): International Conference on Web Engineering, ICWE 2003, Proceedings. Lecture Notes in Computer Science 2722, Springer (2003) 30–39.
8. Iglesias, C., Mercedes Garijo, M., Gonzalez, J. C., and Velasco, J. R.: *Analysis and design of multiagent systems using MAS-CommonKADS*. In: Singh, M.P., Rao, A., Wooldridge, M.J., (eds.): Intelligent Agents IV. LNAI 1365, Springer-Verlag (1998) 313–326.
9. INGENIAS IDE, <http://sourceforge.net/projects/ingenias/>.
10. Kuutti, K.: *Activity Theory as a potential framework for Human-computer interaction research*. In B.A. Nardi, (ed.), context and consciousness: Activity Theory and Human-Computer Interaction. Cambridge, MA: MIT press. 1996.
11. Leontiev, A. N.: *Activity, Consciousness, and Personality*. Prentice-Hall. 1978.
12. Maes, P.: *Modeling Adaptive Autonomous Agents*. Artificial Life Journal 1, No. 1 & 2, MIT Press, 1994.
13. Newell, A.: *The knowledge level*. Artificial Intelligence 18 (1982) 87–127.
14. OMG: *Unified Modeling Language Specification. Version 1.3*. <http://www.omg.org>
15. Pavón J. and Gómez-Sanz, J.: *Agent Oriented Software Engineering with INGENIAS*. In: Vladimír Marík, Jörg Müller, Michal Pechoucek (Eds.): Multi-Agent Systems and Applications III, 3rd International Central and Eastern European Conference on Multi-Agent Systems, CEEMAS 2003, Prague, Czech Republic, June 16-18, 2003, Proceedings. Lecture Notes in Computer Science 2691, Springer (2003) 394–403.
16. Sykara, K.: *Multiagent systems*. AI Magazine 19(2). 1998.
17. Vygotsky, L. S.: *Mind and Society*. Cambridge MA, Harvard University. 1978.
18. Wooldridge, M., Jennings, N. R., and Kinny, D.: The Gaia Methodology for Agent-Oriented Analysis and Design. Journal of Autonomous Agents and Multi-Agent Systems 3 (2000) 285–312..
19. Yu, E.: *Towards Modelling and Reasoning Support for Early-Phase Requirements Engineering*. Proceedings of the 3rd IEEE Int. Symp. on Requirements Engineering (RE'97) Jan. 6-8, 1997, Washington D.C., USA. IEEE Press (1997) 226–235.

A Design Taxonomy of Multi-agent Interactions

H. Van Dyke Parunak¹, Sven Brueckner¹, Mitch Fleischer¹, and James Odell²

¹Altarum, 3520 Green Ct Ste 300
Ann Arbor, MI 48105 USA, +1-734-302-4600
{van.parunak, sven.brueckner, mitch.fleischer}@altarum.org

²James Odell Associates, 3646 Huron River Drive
Ann Arbor, MI 48103, +1-734-994-0833
email@jamesodell.com

Abstract. Agent interactions are frequently characterized as “coherent,” “collaborative,” “cooperative,” “competitive,” or “coordinated.” These terms specialize the more foundational category of “correlation,” which can be measured by the joint information of a system. “Congruence” is orthogonal to the others, reflecting the degree to which correlation and its specializations satisfy user requirements. A taxonomy of these mechanisms can guide the design of multi-agent interaction. Lack of correlation is sometimes necessary, and requires the use of formal stochasticity.

1 Introduction

Designers of multi-agent systems need to discuss agents’ joint action in a disciplined way that current usage does not support. Some circles emphasize cooperation as the dominant theme (e.g., the title of [23] or the subtitle of [46]), but often the system designer does not care whether the agents cooperate or contend, so long as their behavior is coordinated in a certain way. Market mechanisms achieve coordination through mechanisms that are at least competitive and sometimes contentious. Both contention and cooperation presume a cognitive model that some architectures do not satisfy. Other terms for agent interaction include “coherence” and “collaboration” Taking advantage of the pervasive use of the Latin *co-* and *con-* in the English lexicon, we refer to them collectively (including nominal, verbal, and adjectival forms) as “Co-X,” and from this point capitalize them.

Paper titles from [6, 18], ICMAS’95-00, Agents’97, 98, 00, and 01, and AAMAS’02 illustrate the proliferation of terms describing agent interactions. From 1981 through 1993, “Cooperate”¹ accounted for eight of the twelve “Co-X” terms in titles, “Coordinate” for three, and “Coherent” for one. “Collaborate” appeared in 1994, and “Compete” in 1995. For 1994 through 2002, 120 Co-X terms were used, with “Coordinate” at 41%, followed by “Cooperate” at 38%, “Collaborate” at 16%, and “Compete” and “Cohere” at 2.5% each. The vocabulary is growing beyond “Cooperation.” Yet, review of this literature shows that there is little agreement on

¹ Including the noun “Cooperation” and the adjective “Cooperative.” When we refer to one grammatical form of a given word, we intend our observations to apply to the others as well.

defining the various members of Co-X. Formal analyses of teams and social behavior (e.g., [11, 12, 13, 43]) helpfully refine concepts such as common knowledge, joint intentions, commitments, obligations, and social norms. These and related concepts help achieve the behavior that Co-X describes, but the usage of the Co-X terms themselves remains intuitive and sometimes inconsistent.

A taxonomy of these and other terms can enable a more precise description of how agents interact. This precision is a critical contribution to agent-oriented software engineering, since it provides a foundation for improved system specifications and improved communication among users, designers, and implementers of multi-agent systems. Under our definitions, the terms are neither a mutually exclusive spanning set such that every agent-based system belongs to exactly one term in the set (“categories”) nor an orthogonal set each of whose terms can be applied to all agent-based systems (“perspectives”). A formal taxonomy requires a complete structure of both categories and perspectives [31], but at this point we claim only, in the words of one reader of an earlier draft, “a nice start.”

Section 2 defines the most general term, “Correlation,” statistically, without assumptions about either the internal structure of the agents or the relative centralization or decentralization of their behavior. Section 3 defines “Coordination” as Correlation that results from information flows among agents. When these flows result from individual agent intentions, we speak of “Cooperation” and “Competition,” discussed in Section 4, along with “Collaboration,” which is the intersection of Cooperation and “Conversation” (a specialization of Coordination). Section 5 discusses the “Congruence” of group behavior with system-level intentions. Section 6 returns to the fundamental notion of Correlation and examines ways in which it can and cannot be avoided. Section 7 offers a summary.

2 Correlation: Behavioral Joint Information

The most generic description of what agents do together is their joint information, otherwise known as their correlation entropy, joint entropy, or mutual entropy [2]. This quantity can be determined empirically, without access either to the internal structure of the agents or to the broader system within which they are embedded. A set of agents with positive joint information is “Correlated.” In some cases, such empirical observations may be sufficient to impute cognition to agents whose internal structure is unknown [8, 21, 34].

Joint information can be computed for many aspects of the agents. We are interested in agent behaviors, so we compute it over agent actions. At each time step, each agent (indexed by i) can execute one of n_i actions $\{a_{i1}, a_{i2}, \dots, a_{in_i}\}$. Let p_{ij} be the probability that agent i executes action a_{ij} . We estimate this quantity by maintaining a time series of the agent’s last k actions, counting the actions of each type, and dividing by k . One measure of the agent’s behavior over time is its Shannon entropy,

$H(a_i) = -\sum_{j=1}^{n_i} p_{ij} \log_2 p_{ij}$. This standard definition makes no assumptions about the

independence of the elements in the set that generates the p_{ij} . It is simply an empirical characterization of their relative prevalence. (If k is small compared with agent lifetime, one can index p_{ij} and thus H with time t , referencing the window of length k

centered at t .) We can characterize the entropy of the overall system in terms of the various combinations of actions of individual agents. E.g., for two agents, the maximum total number of system actions is $n_1 * n_2$, p is indexed over joint actions, and the system entropy is $H(a_1, a_2) = - \sum_{j=1}^{n_1 * n_2} p_j \log_2 p_j$ (again, indexable by t).

System entropy is subadditive, $H(a_1, a_2) \leq H(a_1) + H(a_2)$. Equality obtains when individual agent behaviors are statistically independent, and fails when they are dependent. The difference $I(a_1; a_2) \equiv H(a_1) + H(a_2) - H(a_1, a_2)$ is the correlation, mutual, or joint entropy, or the joint information. The latter term avoids connotations of disorder implicit in “entropy.” Agent behaviors in a system with high joint information are more Correlated with one another, and in that sense more orderly, than when joint information is low.

Agents are Correlated when their actions depend statistically on those of other agents. It does not matter at this level whether this Correlation results from information flows among the agents or between them and a central controller, or whether its roots are cognitive or subcognitive. Correlation manifests itself in an increase of the system’s joint information, and we propose using this quantity as a measure of system Correlation. (Thus, one system has a higher Correlation than another just when the joint information of the first is greater than that of the second.) This perspective permits us to distinguish the *fact* of Correlation from the *mechanisms* used to achieve it.

For example, suppose that each of two agents needs access to a widget to perform its duties; that there are two widgets available, and that each agent accesses each widget half of the time. Then the probability that agent a_1 is accessing widget 1 is $p_{1,1} = 0.5$, as are $p_{1,2}$, $p_{2,1}$, and $p_{2,2}$. Assume that a widget works better when only one agent is using it at a time. Each individual agent entropy $H(a_i)$ is $-2 * 0.5 \log_2(0.5) = 1.0$. At the system level, there are four possible joint actions: both agents accessing widget 1, both accessing widget 2, a_1 accessing widget 1 and a_2 widget 2, and *vice versa*. If the agents do not Coordinate their activities, then each of these four possibilities is equally likely, with probability 0.25, and the system entropy $H(a_1, a_2)$ is $-4 * 0.25 \log_2(0.25) = 2 = \sum H(a_i)$, so the joint information is 0. Now assume the agents’ behaviors are Correlated (by whatever means) so that they avoid the joint actions in which they both choose the same widget. Now there are only two system actions, each with probability 0.5, and the system entropy is 1, less than the sum of the agent entropies. The difference, 1, is the joint information between the agents.

We illustrate joint information in a model of multi-agent resource allocation, the minority game [36]. At each time step each of the N agents in the population (where N is odd) seeks to allocate itself to one of two resources. Each agent receives a point each time it reaches the less-occupied (minority) resource. The system goal is to maximize the total points awarded to the entire population (equivalently, to minimize the variance in the population of either resource). The agents have access to a time series identifying which resource was in the minority at each past cycle. In each turn of the game, the agents use the last m entries in this time series to choose the resource they will access on the next time step. The quantity $z = 2^m / N$ reflects the normalized size of the strategy space available to the agents.



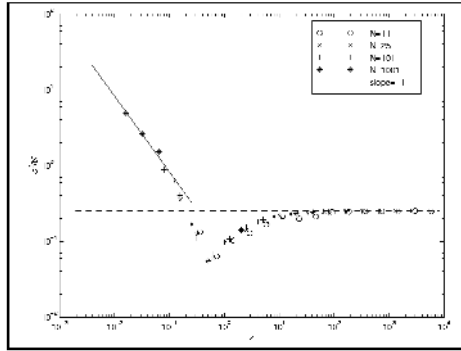


Fig. 1. Inefficiency vs. Size of Strategy Space in Minority Game.

Figure 1 plots the normalized variance σ^2/Nas as a function of z . Low variance reflects high system-level payoff, so desirable behavior is at the minimum of this curve, where $z \approx 0.34$. The dashed line shows the performance if all agents made random choices. The minimum is a spin-glass phase transition, discussed in more detail in [26, 38]. As z decreases below this point, the system performance degenerates until the agents are doing worse than with random choices. In this region, the time series of minority groups [25] show “herding” behavior. Relatively few distinct strategies are available for small m , with high probability that agents’ decisions will overlap. Above the phase transition, the agents do better than random, but as z increases, performance approaches the random limit as an asymptote.

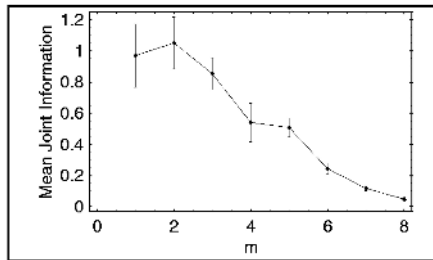


Fig. 2. Joint Information in the Minority Game.—Error bars mark one standard deviation.

Consider a set of 61 agents. Each agent’s entropy is computed from its probability of choosing resource 0 or 1 at each step. The entropy of the system is computed from the probability of a specific vector of 61 individual agent choices at a time step. Thus the size of the system’s state space is $2^{61} \approx 10^{18}$. Reasonable experimental runs with this system are on the order of 10^4 to 10^6 steps, far too short to estimate probabilities over this state space. Instead, we focus on subsystems of six agents each. Such a subsystem has a state space of 2^6 , and we can reliably estimate probabilities with experiments of $10^4 \approx 2^{13}$ steps. Each run of 5001 steps lets us look at ten subsystems of six agents each, and we conduct thirty runs in all. Fig. 2 shows the Correlation, measured by the joint information. This figure has three important features, corresponding to the three regions of Figure 1.

1. The Correlation is highest for low m , consistent with the analysis in [25] showing herding behavior in this region. Within the low region, the mean value appears to increase from $m = 1$ to $m = 2$ before declining for $m > 2$, but given the size of the standard deviation in region, the most that can be said is that the Correlation is comparable for these two values. There are few distinguishable strategies available to the agents for low m , resulting in higher Correlation among their behaviors.
2. High m is associated with low Correlation, corresponding to the region where agent decisions approach the random limit. Thus the general shape of the curve is logistic: low slope for low and high m , and steep slope in between.
3. There is a deviation from this general shape between $m = 4$ and $m = 5$, the region that corresponds to the phase transition (which would be at $m = 4.37$, though our experiments do not sample this point).

This illustration of joint information has several lessons for designers of multi-agent systems.

- Joint information can be computed for a realistic system, offering a basis for metrics that will enable designers to specify degrees of Correlation. Since Correlation is the foundation on which the other members of Co-X are constructed, this formalism is an important tool for comparing different systems in terms of the degree of joint activity that their members achieve.
- When estimating joint information from observations of a system (as opposed to computing it analytically from a system specification), attention must be paid to sampling issues.
- Joint information is not the same as performance. Figure 1, which estimates overall system performance, has a very different shape from Figure 2, which shows Correlation. System designers must recognize this distinction, discussed in more detail in Section 5.

3 Coordination: Communication

Perhaps the most commonly used member of Co-X is “Coordination.” It is prominent in the ACM computing classification system under I.2.11, “Distributed Artificial Intelligence” [1] (paired with “Coherence,” discussed under “Congruence” below). “Correlation” describes simply the fact of statistical non-independence among agent behaviors, while “Coordination” implies a causal process. Correlation can emerge among randomly generated numbers as a statistical fluke, but when it arises from a causal process, that process involves communication, that is, information flow between an individual agent and its environment. The environment, in turn, is everything outside the individual agent’s boundary. The options for this flow stem from the contents of this environment, which include both other agents and environmental state variables. (A side effect of this definition is to require redefinition of either “Coordination” or “communication” in paper titles of the form, “Coordination without communication” [3, 15, 16, 40].)

The environment includes other agents, not only software agents but also human stakeholders, system designers, and conventional computer systems. The relationship between two such agents will be one of two types, depending on the state of the agents and the rules of the system (expressed, for example, in the agents’ roles [35]

Table 1. Categories of Communication

		Topology of Inter-Agent Relationships	
		Centralized (between Distinguished and Subordinate agents)	Decentralized (among Peer agents)
Information Flow	Direct (messages between agents)	Construction (Build-Time) Command (Run-time)	Conversation
	Indirect (non-message interaction)	Constraint	Stigmergy ² (generic) Competition (limited resources)

and the protocols in which they participate). When the agents can say “No” to one another within the rules of the system, they are “peer agents.” When one of them (say agent A) can say “No” to the other (B), but B cannot say “No” to A, we call A the “distinguished agent” and B the “subordinate.” The relationship between two agents may be fairly fixed (for example, the relationship between a human programmer and her software agent). Or it may vary over time (as when peer agents negotiate a work plan that calls for one of them to supervise the other, resulting in a distinguished-subordinate relationship during execution). These concepts can be developed more formally through dependency and autonomy theory [10, 30].

The environment has state variables that the agents can sense, and may support its own processes that couple its state variables and cause them to change over time. Environmental variables are of two types. The values of *endogenous* variables change over time depending on the actions of peer agents. The values of *exogenous* variables, such as sunspot frequency, change independent of the actions of peer agents, but may result from actions of the distinguished agent, and are often scripted by the system designer.

In such an environment, Correlation mechanisms can be either centralized or decentralized. In addition, the information flows involved may be either direct from agent to agent (ignoring those aspects of the environment that make up the communication system) or indirect (mediated through explicitly modeled environmental state variables). Table 1 reflects these categories, which offer a design template for agent interaction.

Centralized mechanisms for Correlation all involve communication between the distinguished agent and its subordinates. This flow may be direct (when the distinguished agent Constructs or Commands the subordinates) or indirect (when the distinguished agent Constrains the subordinates by manipulating exogenous environmental variables visible to the subordinates). In Correlation through Command, used commonly in robot soccer, holonic manufacturing, and some simulation applications, agents behave much like objects, executing methods invoked by incoming messages. The focal point algorithm advocated by [15] and the common utility functions implicit in [16] both rely on Construction (common programming). In indirect centralized mechanisms, subordinates jointly sense changes in a shared exogenous environmental variable. The variable’s dynamics are independent of agent actions, so it cannot move information between subordinates. But it may serve as a synchronizing signal that Correlates the agents’ actions. The experimenter who configures targets and obstacles in an experimental testbed is Constraining the subordinates, supporting Correlation through indirect centralized action.



Decentralized mechanisms for Correlation all involve communication among peers. Most negotiation research focuses on direct peer-to-peer information flows (“Conversation”). Indirect decentralized flows occur when peers make and sense changes to endogenous environmental variables. This class of Coordination is called “stigmergy,” [17], from the Greek words *stigma* “sign” and *ergon* “work”: the work performed by agents in the environment guides their later actions. Such techniques are common in biological distributed decentralized systems such as insect colonies [32]. A common form of stigmergy is resource Competition, which occurs when agents seek access to limited resources. For example, if one agent consumes part of a shared resource, other agents accessing that resource will observe its reduced availability, and may modify their behavior accordingly. Even less directly, if one agent increases its use of resource A, thereby increasing its maintenance requirements, the loading on maintenance resource B may increase, decreasing its availability to other agents who would like to access B directly. In the latter case, environmental processes contribute to the dynamics of the state variables involved. (We reserve “Competition” for resource Competition as a subset of Stigmergy. For the more generic opposite of “Cooperation,” we prefer “Contention,” discussed below.)

We became aware of the active role that the environment plays in negotiation when experimenting with an instance of the contract net for manufacturing control [29]. After carefully proving that our protocol was deadlock-free, we ran it on a physical control system, and it promptly deadlocked. The negotiation in question concerned the movement of a physical part from one workstation to another. Our analysis of the protocol neglected the movement of the physical part itself. This movement conveyed information between the two workstations, and thus between the software agents that represented them. It represented an undocumented extension of our protocol, one that invalidated our proof and caused the system to deadlock. The arrival of the part at the receiving workstation gave that workstation information about the state of the system that it would not otherwise have had, namely, that the part had been delivered.

Traditionally, the study of negotiation focuses on Coordination by means of information flow directly from one agent to another. The mantra of situated robotics that “the world is its own best model” [7] suggests that the problem domain may deserve a more prominent role in the process. There are several motives for understanding the role of the environment in Coordination, and learning to exploit it where possible.

- It supports open, heterogeneous societies of agents. The environment is by definition accessible to the agents that are negotiating about it. Any agent that wishes to deal with the domain must be able to sense and manipulate it. Thus the physics of the environment define common standards for agent interaction, in contrast with the more arbitrary standards programmers can impose on direct agent-to-agent communication.
- It integrates and reflects the state and dynamics of the overall problem-solving process at a global level that is only imperfectly visible in any individual agent’s internal model. In particular, it captures high-order interaction effects that may escape the notice of any individual agent or *a priori* model maintained by an individual agent. For instance, assume agents A, B, C, and D are all interested in resource κ , but A and B know only of each other, as do C and D. The load on resource κ integrates information about the demands of all the agents that would otherwise not be available to them.

- It embeds domain constraints (e.g., resource limitations) directly in the reasoning process, without the need to identify and model them in advance.

A stock market illustrates the importance of information flows mediated by endogenous environmental variables. It affects both stock traders and business executives, in different ways. Traders (at least those who obey SEC regulations) do not communicate directly to determine which shares each will buy and sell. But when a trader offers for sale a share in one company, the offer tends to depress that company's share price, making the company more attractive to potential buyers. Thus information flows between traders through the stock market without Conversation. In contrast, business executives rely extensively on Conversation in reaching contracts with their customers and suppliers. However, they must also pay attention to indirect information flows, including those through the same stock market. For example, if a supplier's stock price drops precipitously, the supplier may not be able to raise needed capital, and in spite of its explicit promises in a negotiation, it may not be able to fulfill its obligations.

The Minority Game is an excellent example of Stigmergy, and of Competition in particular, and we discuss its implication for indirect Coordination further in [36].

These mechanisms reflect Coordination mechanisms recognized by sociologists in organizational design. One prominent discussion [27] distinguishes five such mechanisms:

1. Mutual adjustment, informal communication among workers, corresponds to the Direct Decentralized quadrant of Table 1, which we call "Conversation."
2. Direct supervision is our "Command," which represents the real-time portion of our "Direct Centralized" quadrant.
3. Standardization of work processes (e.g., setting up a work station on an assembly line) is adjusting the environment to Constrain agents to behave in a certain way, and thus corresponds to our "Indirect Centralized" quadrant.
4. Standardization of outputs insures that intermediate outputs from one worker can be input to the next, enabling "stigmergy," our "Indirect Decentralized" quadrant.
5. Standardization of skills and knowledge trains workers to behave in Coordinated ways. This is our "Construction," in the "Direct Centralized" quadrant of the table.

4 Cooperation² and Contention: Intent

Correlation is an empirical concept that requires no knowledge about agents' internal structure or outward organization. The focus on communication emphasized by *Coordination* requires attention to inter-agent issues, but leaves agents' internal logic undefined. Cooperation and Contention involve the agents' intentions. For example, behaviors of traders in a commodity market are highly Correlated, resulting from information flows among them (thus Coordination). Are they Cooperating or Contending? Two traders bidding for the same commodity might be Contending (each seeking to wrest control from the other), Cooperating (pumping the price up to

² Axtell [4] notes that game theoreticians would reverse our definitions of "coordination" and "cooperation." Our definitions are more in line with the usage in the MAS community. The exact words used are much less important than precision in distinguishing the processes involved.

increase the value of their current holdings), or simply Competing (in the sense defined in Section 3). The difference can only be resolved by determining their intent (compare [24]).

A necessary condition for Cooperation is the existence across the Cooperating agents of joint intentions (e.g., [11]). Similarly, contention suggests one agent's intention to frustrate another. To our knowledge, this notion of "antagonistic intent" has not been formalized, but could be along the same lines as joint intention. We do not require intention for Competition, respecting the common use of the term for agents seeking common limited resources without harboring malice toward one another.

The definition of Cooperation and Contention as Correlation driven by agent intent has two important implications.

First, imputing Cooperation and Contention to agents also requires imputing cognition to them. This requirement is easy for Cognitive architectures (e.g., SOAR [28] or BDI [37]), which imitate human cognition. It is less direct for Behavioristic agents. Such agents, inspired by work in artificial life [33], are "black boxes," defined only by their outward behavior, and their internal programming makes no claims to imitate cognition. It is possible to impute cognition to such agents in a disciplined way [34], but doing so would require observing other behaviors beyond the specific actions to be classified as Cooperation or Contention, in order to deduce their intentions toward one another.

Second, neither Cooperation nor Contention requires direct decentralized communication ("Conversation," Table 1), but might result from centralized design-time information flows or interactions through the environment. The special case when agents both Converse and Cooperate is "Collaboration"; "Coalition" describes the resulting state of affairs.

5 Congruence and Coherence: Usefulness

None of the modes of interaction discussed thus far is necessarily desirable. Consider the simple problem of Correlated access by two agents to two widgets considered in Section 1. If the agents avoid choosing the same widget, they are Correlated, achieving a joint information of 1. They will be just as Correlated if they always choose the same widget, but in this case productivity would be lower in the Correlated system than in the random one. Similar examples can be constructed to illustrate that increased Coordination, Cooperation, and Competition do not always result in more productive systems.

The crucial insight here, and one of great importance for agent-oriented software engineers, is that systems can be associated with goals at two levels: the system, and the individual agents. (Cognitive agents reason explicitly about these goals, while in behavioristic agents they are imposed by the agent designer, but they are still agent-level goals.) Categories such as Contention and Cooperation take into account individual agent goals, but not system goals. We propose "Congruence" to characterize the degree to which the pattern of agent interactions (at any level from Correlation through Contention and Cooperation) satisfies ("is Congruent with") system-level goals. "System-level" is critical. For example, in an e-commerce system, each individual agent may have a different user with different goals (e.g., increased

market share vs. short-term profit). Congruence deals, not with the conformity of individual agents to the goals of their respective users, but with the conformity of the system as a whole to its system-level goals (e.g., bounded transaction times, information availability, and transaction security). The relation among the agents that yields Congruence is “Coherence” (Figure 3), a term that without definition in the ACM Computing Classification [1]. (Durfee et al. [14] define “Coherent” as “well-coordinated.”)

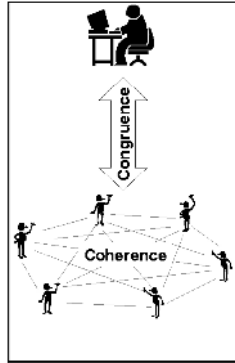


Fig. 3. Congruence and Coherence

System-level goals can arise in two different ways. In an engineered system, they are defined by the system’s creators [41]. We term these “top-down goals.” In a cognitive multi-agent system, they can also emerge from agent interactions [42], whether through democratic processes or by the imposition on other agents of the individual goals of an agent that has gained a controlling position in the society. These goals are thus “emergent goals.” Several points need to be made.

1. Congruence does not presuppose either peer-to-peer information flows or individual agent intent. It may exist with any form of Correlation.
2. Congruence cannot be defined for systems that do not have system goals. Like intentions, emergent goals are most naturally associated with cognitive agents. Behavioristic agents can be congruent in three ways. First, their creator may define their system-level goal. Second, in a larger system that includes both non-cognitive and cognitive agents, the cognitive agents may define emergent system goals for the entire system, and do their best to impose them on the non-cognitive portion of the system. Third, such goals may be imputed to them per [34].
3. A system may have conflicting goals, arising in populations of non-cognitive agents from inconsistency in the designer’s goals, in populations of cognitive agents from tensions among different emergent processes, and in created systems of cognitive agents from a disjunction between the designer’s goals and goals that emerge within the population. Congruence is defined only with respect to a specified goal or set of goals.

System-level goals are needed to *define* Congruence, but whether or not they *affect* it depends on whether individual agents can sense and respond to them. Emergent goals that do change the behavior of the system exemplify “downward causation” [39].

The minority game exemplifies a top-down system goal (maximum total points awarded across the population). This goal is not downwardly causative, because the

agents do not know of it or reason about it, and the system is Congruent only in the vicinity of the phase transition. Changes in Correlation (Figure 2) reflect the coarse structure of system performance (Figure 1), but are not statistically correlated with them. In particular, the highest level of Correlation (and thus Coordination) occurs for low m , while the system is most Congruent and the agents most Coherent for intermediate levels of m .

System-level goals (e.g., “norms,” “conventions,” and “obligations”) are the subject of considerable study ([13] and references). Whatever such theory one adopts will set a standard against which to assess Congruence.

6 Anticorrelation

The various members of Co-X are all refinements of Correlation. Our discussion of Congruence and Coherence suggests that more Correlation is not always good thing.

1. The agent system might be in Contention with an adversary that could exploit observed regularities in its performance. In such cases, the system should avoid regularities, and appear as though it were made up of statistically independent entities.
2. Due perhaps to similarities in their internal coding, the agents may tend to “run into” each other in the problem space, and need to spread out to do their job effectively.
3. The agent system may use some form of weak search (e.g., particle swarm optimization [22] or evolutionary computation [19]) to search the space of system behaviors. Such mechanisms assume ergodicity: they depend on the system’s dynamics to sample the state space, and Correlation leads to under-represented regions of the state space.

Agents can anticorrelate if each makes its decisions via random processes (either at a central controller or in each agent). Can they be more deliberate, basing decisions on either peer-to-peer or master-slave information flows? We can make two observations.

1. Any system with at least two Correlated agents is Correlated. Let $A = \{a_i\}$ be the set of peers, $B = \{b_i\} \subset A$ the subset that is Correlated, $H(A)$ the entropy of the entire system, and $H(a_i)$ the entropy of the i th peer. Perfect anticorrelation requires $H(A) = \sum H(a_i)$ (summing over the elements of A). A component of this sum is $H(B)$. But B is Correlated, so $H(B) < \sum H(b_i)$. Each b_i contributes less than $H(b_i)$ to $H(B)$ and thus to $H(A)$, so $H(A) < \sum H(a_i)$, and the entire system is Correlated.
2. Any set of more than one anticorrelating agents must use a random process in their decision-making to achieve anticorrelation. Assume otherwise. Then their actions are a deterministic function either of a non-random central signal or of observations (direct or indirect) of one another’s behavior. But then each agent’s behavior is not statistically independent of the actions of the other agents, and the system will be Correlated.

Observation 1 makes it unlikely that MAS engineers will ever deal with perfectly anticorrelated systems. Correlation wants to happen. If agents are behaving in any way other than randomly, their aggregate behavior will reflect it. In other words, emergent behavior is ubiquitous. This behavior may not be Congruent (e.g., herding

in financial markets), but it will be Correlated. Rather than viewing emergent behavior as a threat to be suppressed by constraining the behavior of individual agents so that the system exhibits only a subset of its total potential behavior [9, 20, 45], software engineers should understand the mechanisms that drive emergence so that we can harness it for productive use.

Observation 2 emphasizes the importance of stochasticity as an element of multi-agent systems. If we want agents to spread out through their joint state space, we can do no better than to have them flip coins. In the parlance of statistical mechanics, such a device provides the “symmetry breaking” that avoids undesirable Correlation. A system’s level of organization is inversely proportional to its level of symmetry [5], and random variations among agents is a powerful way to introduce differences that can be amplified by agent interactions to yield self-organization. Many techniques of swarm intelligence [32] include a stochastic element, including random walks in ant path planning, Fermi functions in swarm sorting algorithms, and stochastic face-offs in the emergence of organization in *Polistes* wasps. Elsewhere [34] we exhibit a simple artificial agent whose performance is dramatically improved by addition of random noise.

7 Conclusion

Agents do things together. Clear discussions of what they do, and effective designs of how to do it, require precision in the terms we use to describe joint behavior. Such improved precision is particularly critical for software engineers responsible for specifying, designing, constructing, and deploying multi-agent systems.

The fundamental characteristic is *Correlation*, defined as nonzero joint information over a population of agents. Agent Correlation is a purely behavioral notion. It requires knowledge only of the observed actions of the agents. If we admit other sorts of knowledge, we can refine it in three orthogonal ways.

Coordination is Correlation with a focus on the information flow that enables it, and six different flavors can be distinguished: Conversation, Construction, Command, Constraint, Stigmergy, and Competition. The main distinctions are whether the information flow is centralized or peer-to-peer, and direct or indirect. Thus Coordination implies a particular architecture between agents, but is silent about their internal processing.

Cooperation and *contention* modulate Correlation by the intent of individual agents. Cooperation requires joint intentions, while Contention requires an intention on the part of one agent to frustrate another. Both concepts impute cognition to the participating agents (thus requiring special care in the case of behavioristic agents), but they are silent regarding the inter-agent architecture, and thus independent of Coordination. A system with both Conversation (direct peer-peer communications) and Cooperation (joint intent on the part of the individual agents) exhibits *Collaboration*, which results in *Coalitions* of agents.

Congruence measures alignment of an agent system with a system-level goal, which may be defined either endogenously or exogenously. It is independent of both inter-agent and intra-agent architecture. *Coherence* is the relation among agents that yields Congruence. Congruence is not necessarily a monotonic function of

Correlation. Sometimes increased Correlation (or Coordination, or Cooperation) may yield lower Congruence.

More disciplined attention to these distinctions will enable more effective specification, design, and deployment of multi-agent systems. This analysis suggests a number of directions for future work.

- This preliminary taxonomy should be extended. For example, it is fruitful to consider temporal distinctions in the ways agents work together, such as synchrony vs. asynchrony [44].
- The taxonomy provides a basis on which to review current agent engineering models and modeling languages (e.g., AUML) for completeness and expressivity.³
- The next step after defining the members of Co-X is to discuss how to design agents that achieve them. Such a discussion could extend the preliminary suggestions in this paper (e.g., our taxonomy of communication types) with specific patterns and criteria for selecting among them.

Acknowledgements. This work is partly supported by DARPA ANTS and NA3TIVE under contracts F30602-99-C-0202 and N00014-02-C-0458 to Altarum. The views and conclusions in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

References

- [1] ACM. The ACM Computing Classification System [1998 Version]. 1998. HTML, <http://www.acm.org/class/1998/>.
- [2] C. Adami. *Introduction to Artificial Life*. New York, NY, Springer Telos, 1998.
- [3] R. C. Arkin. Cooperation without Communication: Multiagent Schema-Based Robot Navigation. *Journal of Robotic Systems*, 9(3):351–364, 1992.
- [4] R. Axtell. Personal communication 2002.
- [5] P. Ball. *The Self-Made Tapestry: Pattern Formation in Nature*. Princeton, NJ, Princeton University Press, 1996.
- [6] A. H. Bond and L. Gasser, Editors. *Readings in Distributed Artificial Intelligence*. San Mateo, CA, Morgan Kaufmann, 1988.
- [7] R. A. Brooks. Intelligence Without Representation. *Artificial Intelligence*, 47:139–59, 1991.
- [8] B. Browning, G. A. Kaminka, and M. M. Veloso. Principled Monitoring of Distributed Agents for Detection of Coordination Failures. In *Proceedings of Distributed Autonomous Robotic Systems (DARS-02)*, 2002.
- [9] S. Bussmann. Agent-Oriented Programming of Manufacturing Control Tasks. In *Proceedings of Third International Conference on Multi-Agent Systems (ICMAS'98)*, 57–63, IEEE Computer Society, 1998.
- [10] C. Castelfranchi. Founding Agent's 'Autonomy' on Dependence Theory. In *Proceedings of 14th European Conference on Artificial Intelligence*, 353–357, IOS Press, 2000.
- [11] P. Cohen and H. J. Levesque. Teamwork. Technical Report Technote 504, SRI International, Menlo Park, CA, 1991.

³ We are grateful to an anonymous referee for suggesting this and the following point.

- [12] F. Dignum and B. v. Linder. Modelling social agents: Communication as actions. In M. Wooldridge, J. Muller, and N. Jennings, Editors, *Intelligent Agents III*, vol. 1193, *LNAI*, 205–218. Springer-Verlag, New York, NY, 1997.
- [13] F. Dignum, D. Morley, E. A. Sonenberg, and L. Cavedon. Towards Socially Sophisticated BDI Agents. In *Proceedings of Fourth International Conference on MultiAgent Systems (ICMAS'2000)*, 111–118, IEEE Computer Society, 2000.
- [14] E. H. Durfee, V. R. Lesser, and D. D. Corkill. Coherent Cooperation among Communicating Problem Solvers. *IEEE Transactions on Computers*, C-36:1275–1291, 1987.
- [15] M. Fenster, S. Kraus, and J. S. Rosenschein. Coordination without Communication: Experimental Validation of Focal Point Techniques. In *Proceedings of International Conference on Multi-Agent Systems (ICMAS'95)*, 102–108, AAAI, 1995.
- [16] M. R. Genesereth, M. Ginsburg, and J. S. Rosenschein. Cooperation without Communication. In *Proceedings of National Conference on Artificial Intelligence (AAAI'86)*, 51–57, AAAI, 1986.
- [17] P.-P. Grassé. La Reconstruction du nid et les Coordinations Inter-Individuelles chez *Bellicositermes Natalensis* et *Cubitermes* sp. La théorie de la Stigmergie: Essai d'interprétation du Comportement des Termites Constructeurs. *Insectes Sociaux*, 6:41–84, 1959.
- [18] M. N. Huhns and M. P. Singh, Editors. *Readings in Agents*. San Francisco, CA, Morgan Kaufmann, 1998.
- [19] C. Jacob. *Illustrating Evolutionary Computation With Mathematica*. San Francisco, Morgan Kaufmann, 2001.
- [20] N. R. Jennings. On Agent-Based Software Engineering. *Artificial Intelligence*, 117:277–296, 2000.
- [21] G. A. Kaminka, M. Fidanboylyu, A. Chang, and M. Veloso. Learning the Sequential Behavior of Teams from Observations. In *Proceedings of RoboCup Symposium*, 2002.
- [22] J. Kennedy, R. C. Eberhart, and Y. Shi. *Swarm Intelligence*. San Francisco, Morgan Kaufmann, 2001.
- [23] V. Lesser and D. D. Corkill. Functionally accurate, cooperative distributed systems. *IEEE Transactions on Systems, Man, and Cybernetics*, SMC-11:81–96, 1981.
- [24] A. Lux and D. Steiner. Understanding Cooperation: An Agent's Perspective. In *Proceedings of First International Conference on Multi-Agent Systems (ICMAS'95)*, 261–268, MIT and AAAI, 1995.
- [25] R. Manuca, Y. Li, R. Riolo, and R. Savit. The Structure of Adaptive Competition in Minority Games. Program for the Study of Complex Systems, University of Michigan, Ann Arbor, MI, 1998. URL <http://www.pscs.umich.edu/RESEARCH/pscs-tr.html>.
- [26] M. Marsili, D. Challet, and R. Zecchina. Exact solution of a modied El Farol's bar problem: Efficiency and the role of market impact. 1999. PDF File, http://ttt.lanl.gov/PS_cache/cond-mat/pdf/9908/9908480.pdf.
- [27] H. Mintzberg. *Structure in Fives: Designing Effective Organizations*. Englewood Cliffs, NJ, Prentice-Hall, 1993.
- [28] A. Newell. *Unified Theories of Cognition*. Cambridge, MA, Harvard University Press, 1990.
- [29] H. V. D. Parunak. Manufacturing Experience with the Contract Net. In M. N. Huhns, Editor, *Distributed Artificial Intelligence*, 285–310. Pitman, London, 1987.
- [30] H. V. D. Parunak. Distributed AI and Manufacturing Control: Some Issues and Insights. In Y. Demazeau and J.-P. Müller, Editors, *Decentralized AI*, 81–104. North-Holland, 1990.
- [31] H. V. D. Parunak. Hypercubes Grow on Trees (and Other Observations from the Land of Hypersets). In *Proceedings of The Fifth ACM Conference on Hypertext*, 73–81, ACM, 1993.

- [32] H. V. D. Parunak. 'Go to the Ant': Engineering Principles from Natural Agent Systems. *Annals of Operations Research*, 75:69–101, 1997.
- [33] H. V. D. Parunak, S. Brueckner, and J. Sauter. ERIM's Approach to Fine-Grained Agents. In *Proceedings of NASA/JPL Workshop on Radical Agent Concepts (WRAC'2001)*, Forthcoming, 2001.
- [34] H. V. D. Parunak and S. A. Brueckner. Imputing Agent Cognition from Dynamics. In *Proceedings of Autonomous Agents and Multi-Agent Systems (AAMAS 2003)*, (submitted), 2003.
- [35] H. V. D. Parunak and J. Odell. Representing Social Structures in UML for Agent-Oriented Software Engineering. In *Proceedings of Workshop on Agent-Oriented Software Engineering*, 17–24, 2001.
- [36] H. V. D. Parunak, R. Savit, S. A. Brueckner, and J. Sauter. Experiments in Indirect Negotiation. In *Proceedings of The AAAI Fall 2001 Symposium on Negotiation Methods for Autonomous Cooperative Systems*, 2001.
- [37] A. S. Rao and M. P. Georgeff. Modeling Rational Agents within a BDI Architecture. In *Proceedings of International Conference on Principles of Knowledge Representation and Reasoning (KR-91)*, 473–484, Morgan Kaufman, 1991.
- [38] R. Savit, R. Manuca, and R. Riolo. Adaptive Competition, Market Efficiency, Phase Transitions and Spin-Glasses. PSCS-97-12-001, University of Michigan, Program for the Study of Complex Systems, Ann Arbor, MI, 1997. URL <http://xxx.lanl.gov/abs/adap-org/9712006>.
- [39] R. K. Sawyer. Simulating Emergence and Downward Causation in Small Groups. In *Proceedings of Multi-Agent-Based Simulation (MABS'2000)*, 49-67, Springer, 2000.
- [40] S. Sen, M. Sekaran, and J. Hale. Learning to Coordinate Without Sharing Information. In *Proceedings of National Conference on Artificial Intelligence (AAAI'94)*, 426–431, AAAI, 1994.
- [41] Y. Shoham and M. Tennenholtz. On Social Laws for Artificial Agent Societies: Off-Line Design. *Artificial Intelligence*, 73:231–252, 1995.
- [42] Y. Shoham and M. Tennenholtz. On the Emergence of Social Conventions: modeling, analysis and simulations. *Journal of Artificial Intelligence*, 94(1-2):139–166, 1997.
- [43] G. Tidhar, E. A. Sonenberg, and A. S. Rao. On Team Knowledge and Common Knowledge. In *Proceedings of Third International Conference on Multi-Agent Systems*, 301-308, IEEE Computer Society, 1998.
- [44] D. Weyns and T. Holvoet. Synchronous versus Asynchronous Collaboration in Situated Multi-Agent Systems. In *Proceedings of Autonomous Agents and Multi-Agent Systems (AAMAS 2003)*, (submitted), 2003.
- [45] M. J. Wooldridge and N. R. Jennings. Pitfalls of Agent-Oriented Development. In *Proceedings of 2nd Int. Conf. on Autonomous Agents (Agents-98)*, 385–391, 1998.
- [46] M. Yokoo. *Distributed Constraint Satisfaction: Foundations of Cooperation in Multi-Agent Systems*. Berlin, Springer, 2001.

Automatic Derivation of Agent Interaction Model from Generic Interaction Protocols

José Ghislain Quenum, Aurélien Slodzian, and Samir Aknine

Laboratoire d'Informatique de Paris6,
8 rue du Capitaine Scott,
75015 Paris, France

{jose.quenum,aurelien.slodzian,samir.aknine}@lip6.fr
<http://www.lip6.fr/OASIS/fw.html>

Abstract. Interaction can take place in multi-agent systems (MAS), only if each agent contains features devoted to it. These features are usually difficult to implement and programmers have often to redesign them, even when they take inspiration from generic interaction protocols. Most often, interaction features are not separated from other, functional, ones and this places a limitation on the reusability potential of the agent's interaction capabilities in different contexts. In such a situation inconsistencies are difficult to anticipate and solve.

In this paper, we propose a method to address these issues by making separate but still consistent models for the coordination and the functional aspects of the agents and by deriving automatically the agent's effective interaction model from generic interaction protocols. An implementation of this method is also described.

1 Introduction

Interaction is a key concept in multi-agent systems (MAS). It brings together agents that communicate in order to achieve their goals in the system. But, interaction can only take place if each agent has an interaction model that is consistent with the model the other agents have.

The implementation of sophisticated, dynamic and open agent systems is yet confronted with the complexity of putting together the management of protocols and the functionalities of agents. This burden is an obstacle to the development of multi-protocol agents and hence to the openness of agent systems. Indeed, the external behaviour of agents, even when it is inspired from generic protocols, is most of the time re-programmed by hand.

This situation has two drawbacks. The first is the lack of flexibility. Indeed, any modification on the agent's internal specifications might imply changes on the implementation of the interaction model, and reciprocally. The second drawback is the lack of reusability of the implemented protocols in other contexts and the possibility of inconsistencies at runtime.

We address these issues by defining two different models of the agent, which may be informally defined as follows:

- the *interaction model* of the agent contains a formal description of the sequence of communicative acts the agent commits to in order to respect a number of coordination scenarios.
- the *functional model* of the agent contains a description of the functionality of the agent in terms of information transformation.

Both models are of course related since the elements of the functional model are the behavioural bricks that appear in the interaction model.

We furthermore propose that instead of designing the interaction model by hand, the agent programmer builds it by using the functional model to configure and specialise generic coordination protocols. This approach makes thus the assumption that the agent system contains a public library of generic interaction protocols.

Practically, we propose to the agent designer to start with the design of the functional model of its agent, then to look for possible similarities between the methods it contains and the agent actions which are required by the interaction protocols he wishes to support. Even in case of incompleteness of the the functional model, the programmer has immediately the specification of the missing functionalities to support a given protocol.

On this basis, it is even possible, to assist the programmer by means of a *unification algorithm* which will find out similarities by comparing the data flow descriptions associated to the methods defined in both models. In case of failure we also propose an *adaptation algorithm* which will provide a complete specification of the missing methods. Moreover, when unification is finally successful, the configured protocol may be completely implemented into the agent.

We argue that this methodology, will ease the process of implementing coordination protocols into agents and that, thanks to the unification algorithm will dramatically reduce the the risk of inconsistencies.

Section 2 discusses some related work. Section 3 describes our methodology in details: the nature of the models, the algorithms and their implementation.

2 Related Work

The methodology we propose is related to two fields of multi-agent system design: on the one hand modelling methodologies for the design of MAS and, on the other hand, protocol engineering, which focuses on protocol construction and description.

Many methodologies have been proposed for modelling multi-agent-systems: Gaia [13], Multi-agent Systems Engineering (MaSE) [1], Tropos [4], Prometheus [8], ROADMAP [11], and the Formal Methodology [2] based on UML and graphs transformations, to quote a few. All these methodologies provide means to model the individual agents and their interactions with their environment and hence to produce the agent model, the service model, the environment model and the interaction model, which is based on generic interaction protocols. However, these approaches consider the design of a MAS as a whole while the approach we

propose focuses on making agents interoperable by means of the reuse of generic coordination protocols.

From the protocol engineering perspective, FIPA proposes several generic protocols [3]: FIPA-request (and its variant FIPA-request-when), FIPA-query, FIPA-Contract-net (an extension of the Contract Net Protocol [9]). This group also proposed a UML extension, UAML, to model interactions in MAS. Other UML extensions have been proposed to model agent interactions: UAML_e [5], AUML [7] and EAUML [12].

Mazouzi and al. [6] propose a generic approach for protocol engineering by analysing, specifying and verifying the protocols. This approach derives automatically formal specifications of interaction protocols from semi-formal ones given in protocol diagrams (AUML). This approach emphasises the formal representation of interaction protocols but does not manage the difference between generic and specific protocols.

Tadashige and al. [10] propose a framework for exchange of protocols called *Virtual Private Community*. This framework allows the agents to make their protocols evolve dynamically. It assumes that the protocols are already designed in the agent's model, and provides rules to change the agent's current state according to the interaction execution. It does not focus on the way the protocols are built before starting to evolve dynamically.

The protocol engineering methodologies in fact provide us with techniques of designing new protocols. Our approach is complementary both to this and to MAS methodologies since we are concerned with the integration of the protocols in the agents, and with the consistency of this integration.

3 The Methodology

The methodology we propose is based on the idea of separating the agent's internal specifications from its interaction model, and to generate the later as an instantiation of generic interaction protocols, where the term "instantiation" is to be taken in the sense of a configuration or a specialisation of the generic protocol for the agent. Indeed generic protocols specify the external behaviour of agents only formally: they state what type of messages have to be exchanged and when but they do not say anything about the *contents* of the messages. This is exactly the role of this configuration.

The relationship between the internal specifications and the interaction model will be established by means of the *functional model* of the agent, which will contain the description of the methods the agent may use to handle data or to produce information during interactions.

The method we propose is summarised in Figure 1. At the heart of this method are the agent's functional model and a standard library of generic interaction protocols. We then match both models using a *unification algorithm*. This unification identifies the possible similarities between the agent's methods and actions required by the protocol. When all the actions of a role (any communicating entity in a protocol, see below) are not paired with methods, a

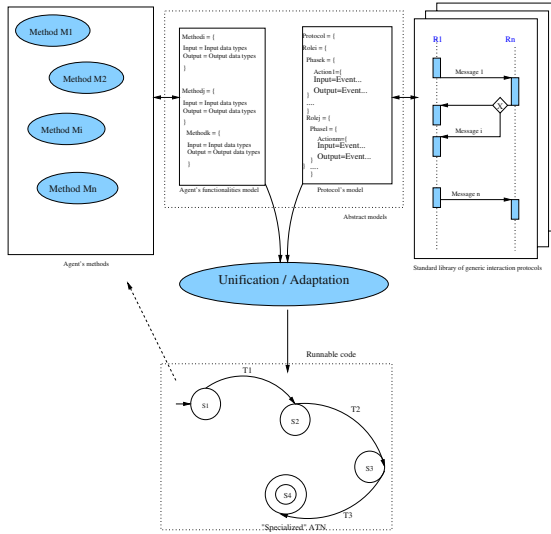


Fig. 1. Interaction models automatic derivation's global schema

reconfiguration of this model is proposed thanks to an *adaptation algorithm*. This reconfiguration consists in identifying the methods that are missing so as to allow the agent to fully perform the desired roles. Finally, when a role is completely unified (all the actions in the role definition are associated with agent methods), then we replace in the interaction model the actions with the corresponding agent methods, and we store the configured role in the agent's interaction model (in the form of a finite state machine).

Our method is organised in four steps:

1. abstract representation of generic interaction protocols, based on UML sequence diagrams;
2. abstract representation of the agent functionalities, describing the methods the agent will use to interact;
3. specialisation of the protocols by matching their actions to the agent's methods;
4. conversion of unified roles into abstract machines, and insertion in the interaction model.

Notice that roles can be selected gradually to run steps 3 and 4.

3.1 The Generic Interaction Protocol

The model that describes the generic interaction protocol uses some concepts that we define first; our definitions here are still informal. We then give an in-depth description of this model and finally provide some representation examples.

Definitions.

Interaction protocol. A context wherein entities are exchanging data or knowledge in order to achieve some goals. It is composed of three elements: roles, messages and partially sorted exchange sequences.

Role. An entity structurally involved in an interaction protocol. The activity of a role is divided into phases. A role is a definition object which imposes constraints on agents that are supposed to play this role.

Phase. A consistent part of an interaction protocol inside a role. While the phases go on, some actions are accomplished so as to cope with the events occurring in the role's environment.

Event. A situation occurring in a role during interactions (phase firing, end of a phase, message reception, message emission, etc.)¹

Action. An operation a role executes in a determined phase. An action might be considered as a method which would have events as input and output arguments.

Once these concepts are defined, an in-depth examination is required to identify the way to combine them.

The graphical representation of sequence diagrams only shows the communication between entities (roles). However, there might be some information the role keeps "centrally" in order to communicate with several other roles in the context of the same protocol. This situation requires us to look for information description beyond the communication aspect, in order to thoroughly describe a role.

Continuing from the concepts and techniques we have just identified, we will build an abstract model that thoroughly describes the role's interactions.

Abstract Model. The abstract model we propose here is formalised as a set of EBNF rules. Here, only a part of the rules are presented and commented. The whole abstract model is described in Appendix D.

1. In this model, a protocol has some properties (**title** and **family**), and is composed of roles and message patterns. Figure 2 shows these elements.

```

<protocol>          → <protocolproperties><roles><messagepatterns>
<protocolproperties> → title family
<roles>             → <role><role>|<roles><role>
<messagepatterns>  → <messagepattern+>

```

Fig. 2. Syntactic rules of a protocol

2. As in Figure 3, a role is decomposed into a set of phases.

¹ cf. table 2 in appendix.


```

<role> → <roleproperties><rolevariables?><actions?><phases>
<roleproperties> → name cardinality
<phases> → <phase+>

```

Fig. 3. Syntactic rules of a role

3. Actions are classified into several types: appending, updating, removing, setting and computing data. Each type has input arguments and/or one output result, all of those being specified by their data types (listed in Table 1). There are also actions that send a message and which, obviously, take this message as an argument. More possible actions consist in stopping a phase or even the whole protocol. Events and message arguments can be combined in lists. The rules describing an action are shown in Figure 4.

```

<action> → <actionproperties><actioncontent>
<actionproperties> → <actiontype> description
<actiontype> → append|custom|remove|send|stopphase|set|update

```

Fig. 4. Syntactic rules of an action

4. A *message pattern* is a set of the message elements that are required to appear in a particular message being exchanged at a particular moment of the protocol. Among those properties one will find the communicative act, the senders, receivers, the message content type, etc. The rules describing a message pattern are shown in Figure 5.

```

<messagepattern> → <messagepatternproperties>
                  <senders><receivers><contenttype><description?>
<messagepattern
properties> → performative currentprotocol

```

Fig. 5. Syntactic rules of a message pattern

Example. The representation format we choose for our model is XML. We use a DTD derived from the EBNF rules we just gave. Let's consider the FIPA-CNP

In this protocol, two roles are defined: the *initiator* and the *participant*. However, for each instance of the protocol there will be one instance of the initiator role interacting with several instances of the participant role.

In this example, three variables, namely `bidlist`, `analysisresult` and `deadline` and one action, called `BidsDeliberation` compose the initiator. The bids are stored in the variable `bidlist`. The variable `analysisresult` contains the result of a bid's analysis. Finally, the variable `deadline` defines when bidding should stop. The action `BidsDeliberation` is to be executed when the

value of the `bidlist` variable changes, which denotes that new bids have been received from participants. This action produces as result a change in the variable `analysisresult`. Note that the notion of variable here is not the same as in a programming language: it is a statement that there should be somewhere some persistent storage where the above mentioned information is stored somehow.

```
<role id="initiator" name="Initiator">
  <rolevariables>
    <variable id="bidlist" type="Collection" name="bidlist"/>
    <variable id="resultvar" type="Boolean" name="analysisresult"/>
    <variable id="deadline" type="Date" name="deadline"/>
  </rolevariables>
  <actions>
    <action type="custom" description="BidsDeliberation">
      <input><event type="change" variable="bidlist"/></input>
      <output><event type="change" variable="resultvar"/></output>
    </action>
  </actions>
  ...
</role>
```

During the first phase, the initiator emits a `cfp` (call for proposal).

```
<phase id="phs1">
  <actions>
    <action type="custom" description="prepareCFP" >
      <input><event type="variablecontent" variable="deadline"/></input>
      <output><event type="messagecontent" message="cfp" id="evt0"/></output>
    </action>
    <action type="send"><message id="cfp"/></action>
    <action type="stopphase"><event type="endphase" id="evt1"/></action>
  </actions>
</phase>
```

After this, the initiator can receive either a `refuse` message, a `notunderstood` message, a `propose` message or no message closing – which closes the whole interaction with the concerned participant.

The description of the protocol continues with the other possible phases of this role and should be completed with the description of the participant role.

3.2 The Agent's Functional Model

Specifications. This functional model of an agent contains the description of the available methods the agent may use to handle or produce information during its interactions. In fact, we do not need that the description of a method contains much more than the description of its arguments and a return value. Indeed, the role of such a description is to allow for a comparison of the methods with the actions described in the interaction models of generic protocols.

Figure 6 shows the syntactic rules of the agent's functional model.

<code><agent></code>	\rightarrow <code><methods><datadefs></code>
<code><methods></code>	\rightarrow <code><method+></code>
<code><method></code>	\rightarrow <code><methodproperties><input?></code> <code><output?></code>
<code><methodproperties></code>	\rightarrow name location

Fig. 6. Syntactic rules of the agent's functionalities

Example. In the example below, two methods are described. The first one, *Deliberation*, has a collection as input and produces a boolean value. It will match the action called *BidsDeliberation* in the interaction model. The second method, *preparingCFP*, takes a date as input and produces an unspecified output. It will match the *prepareCFP* action of the interaction model:

```
<agent>
  <methods>
    <method id="deliberate" name="Deliberation">
      <input><datatype id="collect"/></input>
      <output><datatype id="boolean"/></output>
    </method>
    <method id="generatecfp" name="PreparingCFP">
      <input><datatype id="date"/></input>
      <output><datatype id="gobject"/></output>
    </method>
  </methods>...
  <datadefs>
    <datadef id="informationstring" type="InformationString"/>
    ...
  </datadefs>
</agent>
```

Once the two abstract models are defined, we can match them in order to identify similarities or to enrich the agent's interaction model.

3.3 Matching the Two Models

This part is fundamental in our model. It matches the actions of the protocol and the agent's methods. Actually, only the actions having input and/or output are matched to methods in the agent's model. Algorithms 1 and 2 describe the way we match action and method.

Algorithm 1 Actions - Methods Unification

Require: List L1 (List of actions in the role)

Require: List L2(List of methods in the agent's model)

Ensure: Map m (matching hash table, key : action, value : set of methods)

Ensure: List L (List of unmatched actions)

```
1: for all action ∈ L1 do
2:   for all method ∈ L2 do
3:     if Match(action, method) = True then
4:       insert (action, method) in m
5:     end if
6:   end for
7:   if action not in m then
8:     insert action in L
9:   end if
10: end for
```

To prevent the same method from matching several actions (leading to semantic inconsistencies), we first check whether the method is already paired or not. *MatchOutput* compares the output combination of the action to that of the

Algorithm 2 Match Actions - Method

Require: Object action, method**Ensure:** Boolean (matching result)

```

1: if method not yet paired then
2:   if MatchInput(action, method)=True And MatchOutput(action,
   method)=True then
3:     Return True
4:   else
5:     Return False
6:   end if
7: else
8:   Return False
9: end if

```

method, returning **true** if they are equivalent, and **false** otherwise. *MatchInput* checks if the method's input is a valid instance of the action's input. We distinguish three cases:

✓ **Connector = “and”**: there must be a total equivalence between the input types in the method and the action. The connection is then:

$$Method \equiv Action$$

✓ **Connector = “or”**: at least one type in the action must exist in the method. Conversely, all the types in the method's input must be in the action's input. The expected connection is then:

$$Method \subseteq Action$$

✓ **Connector = “xor”**: all the types used in the method's input must exist in the action. But only one type among all the possible types in the action must be in the method's input. The desired connection is:

$$Method \subset Action$$

Some actions in the interaction model might be left unpaired, because no methods in the agent's functional model match them. An adaptation algorithm (algorithm 3) is then used to generate a description of these actions as methods. This algorithm looks for all the unmatched actions in each of the selected roles and lists corresponding methods to complete the agent's functional model.

Here are the results obtained by unifying the two roles of FIPA-CNP (*initiator* and *participant*) and the agent's functional model.

Figure 7 shows the global result of the unification indicating whether or not all the actions are paired in a role.

Finally, we generate the list of unpaired actions in order to complete the agent's functional model. The resulting list in the case of our example comes as follows:

Algorithm 3 Actions adaptation**Require:** List L1 (List of unpaired operations)**Ensure:** List L (List of generated methods)

- 1: **for all** operation \in L1 **do**
- 2: Create method
- 3: **if** Operation Input Connector = “and” **Or** “or” **then**
- 4: select all the input types in the operation as input type in the method
- 5: **else**
- 6: randomly select an input type in the operation as the type in the method input
- 7: **end if**
- 8: copy the combination of output types in the operation as output types in the method.
- 9: **end for**

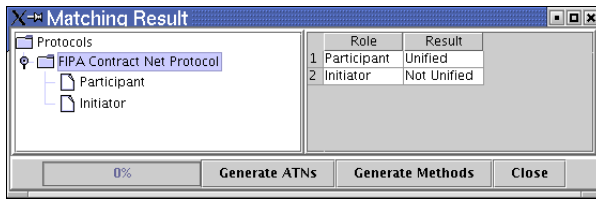


Fig. 7. Roles global unification

```

<agent>
  <methods>
    <method id="mth0" name="singleDeliberation">
      <input><datatype id="boolean"/></input>
      <output><list type="xor"><datatype id="rejectionstring"/><datatype id="acceptancestring"/></list></output>
    </method>
    <method id="mth1" name="handledenial">
      <input><datatype id="denystring"/><datatype id="genericobject"/></input>
    </method>
    <method id="mth2" name="handleProposal">...
  </methods>
  <datadefs>...</datadefs>
</agent>

```

3.4 Final Compilation

The aim of the unification is to identify similarities between actions in the protocols and methods in the agent’s functional model. Once this unification completes, the unified roles are to be specialised (the generic actions will be replaced by corresponding methods) and inserted in the agent’s interaction model. The last step of our method is devoted to this task. Each agent will therefore hold a specialised library of interaction protocols that it will use to participate into interaction scenarios. In our implementation, the role will be implemented as an ATN². In this ATN, each event will make the role change its current state. Actions (whether methods in the case of paired actions or special actions such as those to send a message) related to that event will execute the corresponding transition and make the interaction evolve correctly.

² Augmented Transition Network

4 Conclusion

The method we propose in this paper, offers a way to build the agent's interaction model that will be reusable. By separating the interaction and functional models, there can be flexible updates in the agent's model. The possibility to design a unification algorithm is a guarantee for consistency at runtime.

This method is now going to be applied to two non-trivial application projects, namely Safir (www.projet-safir.org) and Princip (www.princip.net), which both use multi-agent systems to realise sophisticated information retrieval systems. The applicability of the method will then be proved on several tens of interaction protocols.

In particular, we envision that the set of data types we propose might be too limited, and that the type system should allow for inheritance in order to relax some current operational constraints.

Since agents may have many protocols (precisely roles of these protocols) at their disposal, the issue of protocol selection is raised. For the moment it is solved in a rather simple way. We think that since agents dispose of an explicit interaction model, they might be able to select automatically a protocol whether to respond to a received message or to achieve a goal (start a protocol).

Another point we want to improve, is the ability for the agent to dynamically configure its interaction model. In fact, the agent might need, at runtime, to execute a protocol (as initiator or participant) that does not exist in its interaction model yet. Then, the method we describe in this paper should be executed automatically. For this purpose, the agent (the algorithm it uses) is requested to know about the connection between the goals to achieve or task to execute, and the generic versions of interaction protocols the system is provided with, in order to select a set of generic interaction protocols that could be fit.

References

1. S. Deloach and M. Wood. Developing multiagent systems with agenttool. In Springer Verlag, editor, *Proceedings of the 7th International Workshop on Agent theories, architectures and languages*, July 2001.
2. R. Depke, R. Heckel, and J. M. Küster. Formal agent oriented modeling with uml and graph transformation. *Science of Computer Programming*, 2001.
3. FIPA. Fipa interaction protocol library specification. Technical report, Foundation for Intelligent Physical Agents, 2001.
4. F. Giunchiglia, J. Mylopoulos, and A. Perini. The tropos development methodology: Processes, models and diagrams. In *Proceedings of the 1st International Joint Conference on Autonomous Agent and Multi-agents Systems*, 2002.
5. J. L. Koning, G. François, and Y. Demazeau. Formalization and pre-validation for interaction protocols in multi-agents systems. In *Proceedings of the 13th European Conference on Artificial Intelligence*, 1998.
6. H. Mazouzi, A. El Fallah Seghroughni, and S. Haddad. Open protocol design for complex interactions in multi-agent systems. In *Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 517–526, 2002.

7. J. Odell, V. D. Parunak, and B. Bauer. Representing agent interaction protocols in uml. In P. Ciancarini and M. Wooldridge, editors, *Proceedings of the 1st International Workshop on Agent Oriented Software Engineering*, volume 1957. Springer Verlag, June 2000.
8. L. Padgham and M. Winikoff. Prometheus: A methodology for developing intelligent agents. In *Proceedings of the 1st International Joint Conference on Autonomous Agent and Multi-agents Systems*, July 2002.
9. R. G. Smith. The contract net protocol: High-level communication and control in a distributed problem solver. *IEEE Trans. On Computers*, 29(12):1104–1113, 1980.
10. I. Tadashige, W. Yudji, O. Makoto, and A. Makoto. Framework for the exchange and installation of protocols in a multi-agent system. In *Proceedings of the 5th International workshop on Cooperative Information Agents*, pages 211–222, 2001.
11. J. Thomas, A. Pearce, and L. Sterling. Assembling agent oriented software engineering methodologies from features. In *Proceedings of the 1st International Joint Conference on Autonomous Agent and Multi-agents Systems*, 2002.
12. J. Wei, S. C. Cheung, and X. Wang. Towards a methodology for formal design and analysis of agent interaction protocol: An investigation in electronic commerce. In *International Software Engineering Symposium*, March 2001.
13. M. Wooldridge, N. Jennings, and D. Kinny. The gaia methodology for agent-oriented analysis and design. *Autonomous Agents and Multi-Agent Systems*, 3:285–312, 2000.

A Data Types

The types are described in Table 1.

Table 1. Data Types used in the abstract models

Data types and their significance
<i>AcceptanceString</i> : used when the performative is “AcceptProposal”.
<i>AgreementString</i> : used when the performative is “Agree”.
<i>AlwaysRequestObject</i> : used when the performative is “RequestWhenever”.
<i>Boolean</i> : used when the performative is “Confirm”, “Disconfirm”, “Query If”.
<i>BooleanQuery</i> : used when the performative is “Query If”.
<i>BooleanResponse</i> : used when the performative is “Confirm” or “Disconfirm”.
<i>CancellationString</i> : used when the performative is “Cancel”.
<i>ConstrainedRequestObject</i> : used when the performative is “RequestWhen”.
<i>DenyString</i> : used when the performative is “Refuse”.
<i>ErrorString</i> : used when the performative is “Failure”.
<i>GenericObject</i> : used when the performative is “Call for Proposal” or “Propose”.
<i>InformationObject</i> : used when the performative is “Inform Ref”.
<i>InformationString</i> : used when the performative is “Inform-Done”.
<i>NotUnderstoodString</i> : used when the performative is “Not-Understood”.
<i>ObjectQuery</i> : used when the performative is “Query-Ref”.
<i>RejectionString</i> : used when the performative is “Reject-Proposal”.
<i>RequestObject</i> : used when the performative is “Request”.
<i>SubscribeObject</i> : used when the performative is “Subscribe”.

B Event Types

Table 2 describes the different kinds of events that can occur during an interaction.

Table 2. Types of events that can occur during the interaction

Events types and their significance
<i>Change</i> : event happening when the content of some data has changed.
<i>Custom</i> : special event occurring in the system.
<i>Emission</i> : event occurring when a message is sent.
<i>Endphase</i> : event announcing the end of a phase.
<i>Endprotocol</i> : event occurring to close the interaction.
<i>Messagecontent</i> : event indicating that the change occurring is related to a specified message.
<i>Reception</i> : event occurring when a message is received.
<i>Variablecontent</i> : event indicating that the change occurring is related to a variable.

C Action Types

Table 3 describes the actions a role can execute during an interaction.

Table 3. Types of actions that executed during the interactions

Action types and their significance
<i>Append</i> : action used to append data to a collection.
<i>Custom</i> : special action executing calculus during the interactions.
<i>Remove</i> : action used to remove the content of a data.
<i>Send</i> : action used to send a message.
<i>Stopphase</i> : action used to stop the running of a phase.
<i>Set</i> : action used to set the value of a variable.
<i>Update</i> : action used to update the content of a variable.

D Generic Interaction Protocol Formal Model

The formal representation of the interaction protocol is based on a set of syntactic and semantic rules. The syntactic rules are described in table 4.

There are two semantic constraints (static semantics) on the syntactic rules we defined. First, terminal elements *eventmessage* and *eventvariable* can reference message patterns or variables. But both of them can not be referenced at once. To express this semantic rule, let's define the following attributes: **content** for *eventcontent* and **ref** for *messagepattern* and *variable*. Second,

Table 4. Abstract representation of the generic interaction protocol

<protocol>	→ <protocolproperties><roles><messagepatterns>
<protocolproperties>	→ title family
<roles>	→ <role><role> <roles><role>
<role>	→ <roleproperties><rolevariables?><actions?><phases>
<roleproperties>	→ name cardinality
<rolevariables>	→ <variable+>
<variable>	→ name <contenttype>
<actions>	→ <action+>
<phases>	→ <phase+>
<phase>	→ <action+>
<action>	→ <actionproperties><actioncontent>
<actionproperties>	→ <actiontype> description
<actiontype>	→ append custom remove send stopphase set update
<actioncontent>	→ <input?><output?> <message> <event> <eventref> <list>
<input>	→ <event> <eventref> <list>
<output>	→ <event> <eventref> <list>
<event>	→ <eventtype><eventcontent?>
<eventcontent>	→ eventmessage eventvariable
<eventtype>	→ change custom emission ... see Table2
<eventref>	→ eventreference
<message>	→ messagetosend
<list>	→ <listtype><listcontent>
<listtype>	→ and or xor
<listcontent>	→ <event+> <listcontent*> <eventref+> <listcontent*> <message+> <listcontent*> <variableref+> <listcontent*>
<variableref>	→ variablereference
<messagepatterns>	→ <messagepattern+>
<messagepattern>	→ <messagepatternproperties><senders?><receivers?><contenttype>description
<messagepatternproperties>	→ performative<currentprotocol>
<currentprotocol>	→ yes no
<senders>	→ <roleref+>
<receivers>	→ <roleref+>
<roleref>	→ rolereference phasereference
<contenttype>	→ AcceptanceString AgreementString see Table1

terminal elements *eventreference*, *messagetosend*, *variablereference*, *rolereference* and *phasereference* must reference *event*, *messagepattern*, *variable*, *role* and *phase* respectively. Let's define the following attributes: *reference* for *eventref*, *message* and *variableref* and *references* for *roleref*. Let's also extend the previous *ref* attribute to *event*, *variable*, *role* and *phase*. These two semantic rules are described in figure 8.

```

if (<eventcontent>.content = eventmessage) then
  ∃ <messagepattern>, <messagepattern>.ref = eventmessage;
  <eventcontent>.content ≠ eventvariable;
else if (<eventcontent>.content ≠ ∅) then
  ∃ <variable>, <variable>.ref = eventvariable;
  <eventcontent>.content ≠ eventmessage;
end if
1: ∀ <eventref>.reference, ∃ <event>,
  <event>.ref = <eventref>.reference;
2: ∀ <message>.reference, ∃ <messagepattern>,
  <messagepattern>.ref = <message>.reference;
3: ∀ <variableref>.reference, ∃ <variable>,
  <variable>.ref = <variableref>.reference;
4: ∀ {<roleref>.references}, ∃ <role> and <phase>,
  <role>.ref ∈ {<roleref>.references} and
  <phase>.ref ∈ {<roleref>.references};

```

Fig. 8. Semantic rules on generic interaction protocols

E The Functional Model

The agent's functional model is also described by the means of syntactic and semantic rules. The syntactic rules are described in Table 5. Here again, we distinguish two semantic rules. The first rule demands the terminal element *concernedtype* to belong to the set of types declared in *datadef*. The second rule stipulates that a *method* can not lack at the same time the *input*, and *output* elements. Figure 9 describes these rules.

$$\forall \langle \text{datatype} \rangle . \text{reference}, \exists l \in \{ \langle \text{datadef} \rangle . \text{literals} \},$$

$$\langle \text{datatype} \rangle . \text{reference} = l$$

$$\forall \langle \text{method} \rangle, \langle \text{method} \rangle . \text{input} \neq \emptyset \text{ or } \langle \text{method} \rangle . \text{output} \neq \emptyset;$$

Fig. 9. Semantic rules for the agent functional model

Table 5. Abstract representation of the functional model

$\langle \text{agent} \rangle$	$\rightarrow \langle \text{methods} \rangle \langle \text{datadefs} \rangle$
$\langle \text{methods} \rangle$	$\rightarrow \langle \text{method} + \rangle$
$\langle \text{method} \rangle$	$\rightarrow \langle \text{methodproperties} \rangle \langle \text{input?} \rangle \langle \text{output?} \rangle$
$\langle \text{methodproperties} \rangle$	$\rightarrow \text{name location}$
$\langle \text{input} \rangle$	$\rightarrow \langle \text{datatype} + \rangle$
$\langle \text{output} \rangle$	$\rightarrow \langle \text{datatype} \rangle \langle \text{list} \rangle$
$\langle \text{list} \rangle$	$\rightarrow \langle \text{listtype} \rangle \langle \text{datatype} + \rangle \langle \text{list} * \rangle$
$\langle \text{listtype} \rangle$	$\rightarrow \text{and} \text{or} \text{xor}$
$\langle \text{datatype} \rangle$	$\rightarrow \text{concernedtype}$
$\langle \text{datadefs} \rangle$	$\rightarrow \langle \text{datadef} + \rangle$
$\langle \text{datadef} \rangle$	$\rightarrow \text{AcceptanceString} \text{AgreementString}$ see Table1

Building Blocks for Agent Design

Hrishikesh J. Goradia and José M. Vidal

Swearingen Engineering Center
University of South Carolina
Columbia, SC 29208
goradia@enr.sc.edu
vidal@sc.edu
<http://jmvidal.ece.sc.edu>

Abstract. We present our Component-Based Agent Framework, which enables a software engineer to design a set of agents by using a visual component-based toolkit (Sun's BDK), and wiring together desired blocks of functionality. We instantiate this framework in the RoboCup domain by implementing the necessary components. The implementation also serves as a proof of the viability of our framework. Finally, we use this implementation to build sample agents. The proposed framework is a first step towards the merging of agent-based and component-based design tools.

1 Introduction

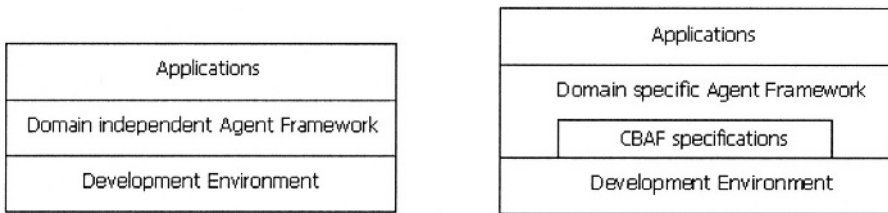
As the Semantic Web [6] and Web services become increasingly ubiquitous we can expect to see an increasing need for agents that can exploit these resources [9]. While some of the agents will be highly complex, we expect that most of them will be simple agents of limited abilities, short lifespan, and built in order to solve temporary problems. Software engineers that have to build these agents will, therefore, want methods that allow them to very quickly develop agents that have the desired capabilities. They will not want to spend time learning to use complex agent architectures in order to build an agent that might only be used a dozen times. They will instead prefer to use the tools that they have grown used to, such as visual component-based development systems.

In this paper, we present our Component-Based Agent Framework (CBAF) along with SoccerBeans—an example application of this framework for the RoboCup [13] domain. CBAF builds on our previous work on an agent architecture for RoboCup [5, 14] by integrating parts of the architecture with the Java Beans component model. The framework is our first step towards the merging of agent-based and component-based design methodologies and tools. Once our framework is instantiated for a particular domain, the resulting system allows the user to build agents using a visual component-based development program (BDK). In this way, the user does not need to think about the domain-independent aspects of building an agent and can instead focus solely on the domain-dependent aspects.

2 The Component-Based Agent Framework Specifications

Despite the recent advances in the quality and the available features in current agent frameworks, designing agent-based systems remains difficult. Both the rigid design specifications of the framework and the necessity to understand the framework’s implementation details have restricted the usability of current frameworks for designing multiagent systems. The CBAF specifications enable a software engineer to design multiagent systems by simply wiring together the desired blocks of functionality for each agent from a pool of available components, thereby circumventing the above-mentioned problems. The frameworks based on the CBAF specifications are simple, generic, and flexible, imposing minimal restrictions and providing a plethora of options for designing agents that will participate in a multiagent system.

The CBAF architecture is a component-oriented approach for designing agent systems. CBAF defines the specifications for a designer to create an agent framework comprising a set of components that can be utilized by the user to develop sophisticated agent behaviors and associate the behaviors with individual agents. Most of the current agent frameworks implement a specific agent system, and a user of these systems is expected to download the provided software and build his agents as extensions to the system. The CBAF approach differs from such frameworks in that it adds a layer of abstraction between the agent system implementation layer and the development environment. The developed agent system will be domain-specific, but by adding this restriction the learning curve for the user, before he can start building applications using the system, becomes virtually non-existent. Figure 1 compares CBAF with other current frameworks. CBAF does not provide any software libraries for agent applications, just a set of rules to be followed by the *designer* while developing agent systems that can be used by the *user* for creating agent applications. We show the usage, benefits, and applicability of the CBAF specifications by presenting an implementation of a soccer simulation system based on the CBAF architecture.



Majority of current Agent Frameworks

Component Based Agent Framework

Fig. 1. Comparison of agent frameworks

2.1 CBAF Specifications

The CBAF specifications defining the necessary environment settings and the rules of agent system design to be followed by the designer and the user of the system are as follows:



- CBAF assumes a discrete, stochastic and episodic environment that can also be dynamic and partially observable with real-time requirements. Also, the set of actions that an agent can take must be bounded.
- CBAF is a component-oriented approach. The development environment selected for designing the agent system must support event-driven programming.
- The agent framework based on the CBAF architecture must be designed as a set of independent, self-contained, highly specialized components. A user must be able to combine the components in various forms to create different plans or activities for the agent. These activities will be associated to the individual agents.
- The agent framework must include the following types of components:
 - *Agent component(s):*
The agent component represents an individual agent in the multiagent system. The component must be able to represent the internal states of the agent and the external states of the surrounding environment. The agent component must also receive sensor input from the environment and produce action output to the environment. The agent will be associated with one or more activity components representing the plans for the agent at design time. The scheduling mechanism for these activities and the internal control flow for the agent must be decided by the agent component. All variations of the above mentioned mechanisms must be represented as separate agent components in the framework.
 - *Activity component(s):*
The activity component is an interface between an agent component and the user-defined activities or plans for the agent. The activity component must fully support the activity scheduling and agent communication mechanisms of the agent component. The activity component must also allow the creation of plans involving complex, decision-making routines. A plan must be rooted by an activity component and generated as a concatenation of rules based on the agent's internal state. The rules can have a positive or a negative classification. For each classification, a new set of rules can be appended. This mechanism produces a data structure like a decision tree. Each branch of the tree structure must be terminated by an action to be performed by the agent when all the rules defined in the path are satisfied. The activity component must be able to solve the plan by propagating events through the plan. The agent framework can have one or more activity components satisfying all the above criteria.
 - *Decision components:*
The decision components represent the rules used in plan generation for an agent. The rules can be defined as individual decision components or as a collection of the components chained together in some order. The framework must include all the decision components, so that the user is able to devise any rule that he wants to use in his plan.
 - *Behavior components:*
The behavior components represent the agent's actions. Each rule set in a plan has to be mapped to a behavior component. The framework must provide every possible atomic action for an agent in the system as a behavior bean.

Figure 2 describes a sample CBAF architecture for the RoboCup domain.

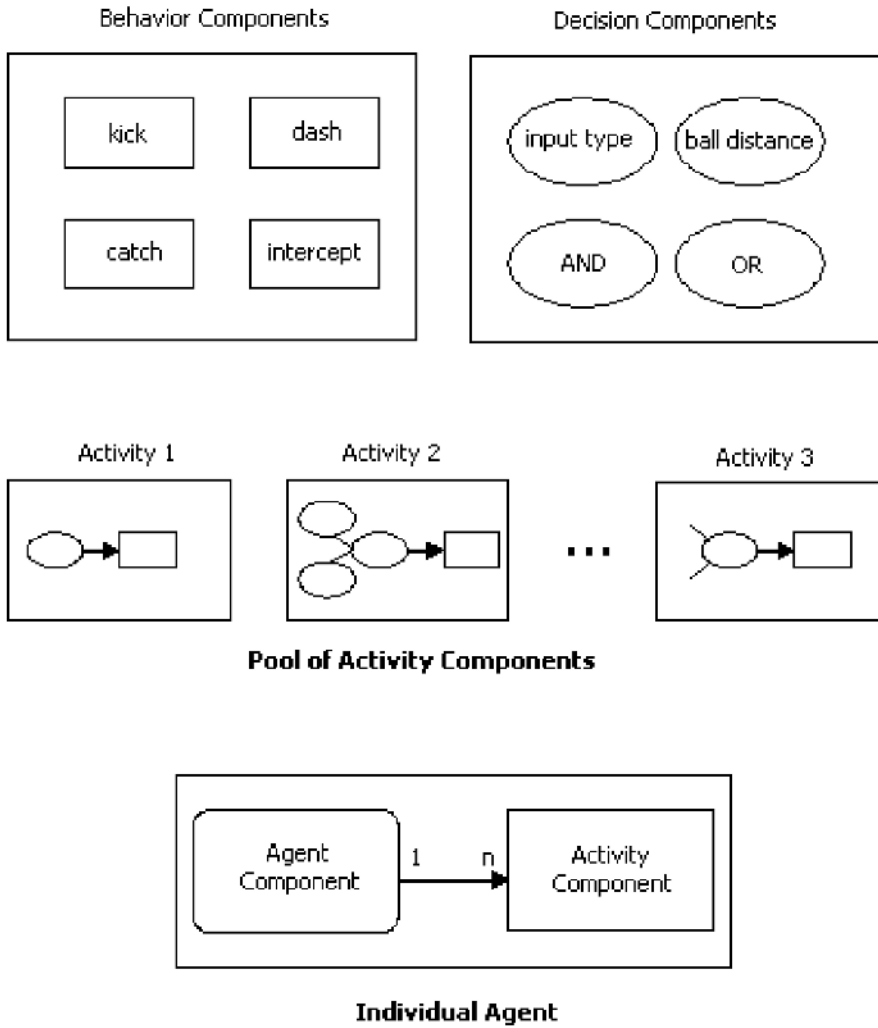


Fig. 2. The CBAF architecture. The sample behavior and decision components are shown for the RoboCup domain

2.2 Challenges for CBAF Design

Some of the issues that need to be addressed for designing the framework described above are:

- Which are the best behavior and decision components to provide to achieve the desired goals?
- The components have to be functionally independent, but they also rely on each other for accomplishing their tasks and hence should be interactive and cohesive. How do we achieve that?

- How should the components be combined to create new components?
- How do we deploy the whole system with multiple agents?

2.3 CBAF Implementation with JavaBeans

The JavaBeans technology [11] by Sun Microsystems seems to be the most suitable development environment for creating agent systems using CBAF. Java supports the *delegation event model* for event handling, where an *event source* generates an event and sends it to a set of *event listeners*. The listeners must be registered with the source to receive notifications about specific event types. JavaBeans leverages the strengths of Java by providing a rich framework for manipulating events and the relationships between event sources and event listeners at design time. The Bean Development Kit (BDK) provided by Sun Microsystems for application development using beans presents a simple way of associating two beans with each other. Linking can be done by simply selecting the method that fires a particular event in the source bean and following it up with selecting the method that needs to be invoked on the listener bean each time that event is fired by the source. The BDK automatically creates the Adapter [7] class for combining the source bean with the listener. Above all, the *introspection* and *reflection* mechanisms supported by Java and extended by JavaBeans open up a plethora of options for aiding complex decision-making by the agents. This paves the way for enabling the creation of a diverse set of applications using the system toolkit. Finally, with JavaBeans, the agent properties can be modified at run-time. This feature can be extremely useful while testing out different strategies for individual agents.

Among the currently available technologies, only JavaBeans provides all of the above-mentioned features. While other development environments like Visual Studio with Visual Basic or Visual C++ can also be used for developing components as DLLs (Dynamic Linked Libraries), they have limitations that add to the complexity of the task of agent system development. While these environments support event handling, combining the event source with the event listener is non-trivial. The Adapter class invoking the appropriate method in the listener component for each event generated by the source component has to be hand-coded. A similar case can also be presented for other distributed computing technologies like CORBA and COM. Since the above-mentioned technologies do not support introspection and reflection, the components will be highly specialized and inflexible. Hence, these technologies would need a much higher number of components to support complex decision-making than JavaBeans. Run-time modifications to agent properties would also be unavailable.

3 The SoccerBeans Framework

SoccerBeans is an implementation of the CBAF specifications for the RoboCup domain. The RoboCup domain is the world of simulated robotic soccer. The SoccerBeans agent system facilitates the creation of different soccer teams with minimal effort on the user's part. SoccerBeans is intended to be used as a pedagogical tool to instigate research in multiagent systems. RoboCup proves to be an excellent test-bed for study and research in multiagent systems' design as it presents a complex, distrib-

uted, real-time, noisy, collaborative, and adversarial environment for extensive research in agent based systems.

The RoboCup simulation system design is based on the client-server architecture. The soccer server provides a virtual field and simulates all movements of a ball and players, and controls a game according to rules. The multiagent system of soccer players forms the client side. Each client controls the movements of one player. Communication between the server and each client is done via UDP/IP sockets.

The SoccerBeans system consists of a pool of functional components developed as JavaBeans (referred to from hereon as just *beans*). The `PlayerFoundation` bean is a highly specialized agent component that represents an individual player in the multi-agent system. The bean handles all the communication, knowledge representation, and activity scheduling aspects of the represented player. The `Activity` bean is the CBAF activity component that glues together the `PlayerFoundation` bean with the different player activities designed from the various decision and behavior beans provided in the system. Eight decision and behavior beans have been developed so far and work is in progress for developing additional beans to further extend the capabilities of the system. As will be shown later in this section, even with such a limited set of components, the CBAF implementation using JavaBeans can support substantially complex decision-making activities.

3.1 PlayerFoundation Bean

The `PlayerFoundation` bean encapsulates all the low level details of the agent and allows the user to concentrate on the real issues of planning and coordination. The `DatagramWrapper` class handles the task of communicating with the soccer server. The player and server configuration information is static and is preserved in the `ConfigurationData` class of the bean. The dynamic information about the surrounding environment received from the server at every simulation step is preserved in the player's `WorldModel`. The bean has a reference to the `RobocupUtilities` class, which is a library of utility functions defining the various actions a player can take. The activity scheduling mechanism of the `PlayerFoundation` bean supports both the BDI and the Subsumption architectures. The design of the `PlayerFoundation` bean is very closely based on the Biter platform [5]. (Please refer to the paper on Biter for further information about the above mentioned classes in the `PlayerFoundation` bean.)

The `PlayerFoundationBeanInfo` class exposes many properties of the `PlayerFoundation` bean. These properties differentiate the players from each other and can be manipulated by the user at design time and, if required, also at run time. The name and team properties define the player's name and team. A value for `playerNumber` is assigned to the player by the server when the initial connection is established. The player's starting position before kickoff and after every goal can be set by the `initialLocation` property. Age defines the number of cycles for which the ghost of a dynamic object is preserved in the world model. The `display` property pops up a new window that graphically displays the player's world model. This can be very useful while debugging. If the `debugFileName` is not empty, then the `PlayerFoundation` bean spits out the debug information into that file. The file defining the player and server configuration information must be entered in the con-

`figFileName` field. `Hostname` and `portNumber` for the soccer server must be defined in the corresponding fields for the bean. The version information is used in establishing the initial connection with the server. This value must be set to 5.00 for the SoccerBeans system. If the player is a goalie, then the `goalie` property must be set to true. The player must be set `online` only after all the other properties are defined and all the activities are added to it. This initiates the socket connection between the server and the player.

As defined by the `PlayerFoundationBeanInfo`, the `PlayerFoundation` bean acts as an event source for the `ActivityEvent` events. The bean provides methods for adding and removing the listeners. The `ActivityEvent` events are fired to pass references of the source bean and the current input to the listeners. The `Activity` beans are the recipients of these event notifications in the SoccerBeans system. While firing these events, the `PlayerFoundation` bean can invoke either of the `addActivity`, `canHandle` or `handle` methods on the listener bean. The `addActivity` method is invoked on an activity at the instant when the activity is registered with the player at design time. The references of both the activity and the player are exchanged at that instant. The `canHandle` method determines if the concerned activity can handle the current input to the player. As part of the `PlayerFoundation` bean's activity scheduling mechanism, the `canHandle` method is invoked for all registered activities at every instant when a new input is received by the player from the server. The `handle` method then executes the concerned activity. Exactly one matching, uninhibited activity is selected for execution at each simulation step.

The `PlayerFoundation` bean is not a recipient of any event notification and hence does not expose any methods for invocation.

3.2 Activity Bean

The design of every new plan or activity for an agent begins with the `Activity` bean. The activities are generated as sets of rules with each branch of the resulting decision tree terminated by an action to be performed by the agent. Thus, if a particular set of rules defined in a path are valid for a player and its surrounding environment then the player performs the action defined at the end of that path. The rules are designed by combining the various decision beans, while the actions performed are defined by the behavior beans. The root of each such path is an `Activity` bean.

The `ActivityBeanInfo` class exposes two properties for the `Activity` bean, the name of the activity, and the `inhibits` property. The list of other activities that are subsumed by this activity must be declared in the `inhibits` property. The activities must be referred to by their name, and delimited by commas.

As defined in the CBAF specifications, the `Activity` bean acts as glue for associating a player to one of its activities. The bean is an `ActionEvent` event listener and implements the `addActivity`, `canHandle` and `handle` methods declared in the `ActivityListener` interface. These methods are exposed for invocation by the `PlayerFoundation` beans through the `ActivityBeanInfo` class. Each activity comprises two decision trees. The tree associated with the `canHandleListener` in the `Activity` bean determines if the current environment settings are favorable for performing the activity. The other tree is associated with the `handleListener` in the

Activity bean and defines how the activity must be executed. The first tree is solved by the `canHandle` method, while the second tree is executed by the `handle` method.

The decision trees in the Activity bean comprise some decision and behavior beans linked with each other in some sequence. For both `canHandle` and `handle`, the Activity bean provides `add` and `remove` methods for decision and behavior beans. The `ActivityBeanInfo` class exposes these methods. The structure of both the methods for solving the trees is identical. A `FunctionalityEvent` event defining the current state of the world is fired from the method and the notification is sent to the decision or behavior bean adjacent to the Activity bean in the chain. Both `DecisionListener` and `BehaviorListener` interfaces listen to `FunctionalityEvent` events and are implemented by the decision beans and the behavior beans respectively. If the next bean in the chain is a decision bean, the Activity bean invokes the `decide` method on the decision bean. If it is a behavior bean, then the `behave` method is invoked. The notification is propagated further in the appropriate path by the decision beans until it hits the behavior bean where it is terminated. For `canHandle`, the behavior bean sets the `canHandle` flag in the Activity bean either directly or via an intermediate decision bean. This determines whether the activity can be performed in the current cycle with the given environment. If the activity is applicable then, as part of the SoccerBeans system's scheduling mechanism, all other activities applicable for the current cycle that are inhibited by this activity are eliminated. The activity that is handled for the current cycle is selected from this new list of applicable plans. For `handle`, the behavior bean typically sends a message to the soccer server describing the player's action for the current cycle.

3.3 Decision Beans

As described in the CBAF specifications, the decision beans enable a user to define rules for generating plans. Every possible condition that the user might need to check for making his/her decision must be encompassed by the published set of decision beans in the framework. The user must be able to define any rule by using either a single decision bean or combining a collection of them in some form. Plans are generated by chaining such rules with an Activity bean at the head and a behavior bean at the tail.

For every path in a plan, a decision bean is preceded by either an Activity bean or another decision bean, and is followed by a behavior bean or another decision bean. Whenever a decision has to be made, a message has to be propagated through a chain from the Activity bean towards the behavior bean via the intermediate decision beans. This is achieved in SoccerBeans by sending `FunctionalityEvent` event notifications through the chain. The event object provides the listener with a picture of the agent's current internal state. Every decision bean implements the `DecisionListener` interface for listening to the `FunctionalityEvent` event notifications. The `decide` method for each decision bean performs the necessary computation and sets the `decision` flag appropriately. The path to be pursued is selected based on the value of the flag, and the event notification is propagated further in that path.

As discussed above, a decision bean can be linked to either another decision bean or a behavior bean. Also, two different chains of beans will be connected to the decision bean, one for each possible value of the `decision` flag. For both chains, a decision bean provides `add` and `remove` listener methods for connecting to both types of beans.

The following decision beans have been developed for the SoccerBeans system to date:

3.3.1 DInputType Bean

The input received from the soccer server is represented as `SensorInput`, while the action event generated by the `PlayerFoundation` bean to elicit an action for the current cycle is represented as `Event` input in the SoccerBeans system. The `DInputType` decision bean can be used to classify player behaviors based on the type of input propagated by the `FunctionalityEvent`. The `DInputTypeBeanInfo` class exposes the `inputType` property of the `DInputType` bean. The valid entries for the property are `SensorInput` and `Event`.

3.3.2 DClosePlayers Bean

The `DClosePlayers` bean can be used to create rules based on the number of other players within some distance to the player represented by the `PlayerFoundation` bean. The players considered can be from either team or irrespective of the team. A user can also set the considered distance and the number of players considered at design time. The `DClosePlayersBeanInfo` class exposes the properties `teamName`, `distance` and `number` for the above functions. The `DClosePlayers` bean executes the `playersInCone` method defined in the `RobocupUtilities` class to determine the result. The decision flag is set to `true` if the number of players counted is less than the value of the number property, else it is set to `false`.

3.3.3 DBallDistance Bean

The `DBallDistance` bean can be used for classifications based on the distance of the ball from the player. The `DBallDistanceBeanInfo` class exposes the `DBallDistance`'s `distance` property, which allows a user to specify the threshold distance. The actual distance is determined from the player's world model. The decision flag is set to `true` if the determined distance is less than the value of the distance property, else it is set to `false`.

3.3.4 DIfThenElse Bean

The `DIfThenElse` bean is defined for classification based on a rule for which there is no explicitly defined decision bean. The `DIfThenElse` bean can be linked to any behavior bean or a chain of decision beans terminated by a behavior bean for defining the rule. The `DIfThenElse` bean passes the `FunctionalityEvent` event notification to the linked bean for determining the decision. The terminating behavior bean must set the decision flag of the `DIfThenElse` bean to `true` or `false`. The `DIfThenElseBeanInfo` class provides an additional set of `add` and `remove` listener methods for the new chain.

3.4 Behavior Beans

As described in the previous sections, plans are generated as sets of rules, with each branch of the resulting tree structure terminated by an action to be performed by the agent. These actions are defined as behavior beans. The actions typically include setting a particular property of some object or executing a particular method for some object. The agent framework must comprise all the behavior beans such that a user is provided with the opportunity to perform any action deemed suitable for his/her plan. The behavior beans are preceded by either an `Activity` bean or a decision bean. Whenever a decision is to be made in SoccerBeans, a behavior bean receives a `FunctionalityEvent` event notification from the preceding bean. Every behavior bean is defined as a listener of these events by implementing the `BehaviorListener` interface. The `behave` method for each behavior bean performs the specified action for the bean.

The following behavior beans have been developed for the SoccerBeans system to date:

3.4.1 BBoolean Bean

The `BBoolean` bean can be used for setting a particular boolean property of an object to the specified value. The `BBooleanBeanInfo` class exposes the `className`, `propertyName` and `value` fields of the `BBoolean` bean. These fields can be used to specify the property that needs to be modified with its new value and the class defining the property. The bean uses Java's Reflection API to locate the set method for the property in the specified object and invokes the method, passing the specified value as the parameter. Thus, any boolean variable in the system can be modified at run time by using this bean.

3.4.2 BIncorporateObservation Bean

The `BIncorporateObservation` bean is meant for receiving the sensor inputs from the soccer server and incorporating them into the player's world model. There is no property in the bean for a user to modify and the `BIncorporateObservationBeanInfo` class is defined accordingly.

3.4.3 BDribbleBallToPoint Bean

The `BDribbleBallToPoint` bean allows the player to dribble the ball to the specified point on the field. A user can specify the point at design time through the `finalPosition` property. This feature can be useful for testing purposes, but has limited applicability at run time as the desired final position for the ball in the soccer game will almost always vary at different points in time. The user would rather like to execute some method for computing the desired final position for the ball at a given time. The `BDribbleBallToPoint` bean provides this option by presenting other fields to the user at design time. The `BDribbleBallToPointBeanInfo` class also exposes the `method`, `className`, `methodName` and `methodParams` properties of the

bean. The `className`, `methodName` and `methodParams` properties can be used to specify the appropriate method with parameters that should be executed for computing the desired final position of the ball. The method is invoked at runtime using the Reflection API. The `method` flag is used to determine whether the ball's final position must be computed by executing the specified method or by considering the value of the `finalPosition` field. The bean executes the `dribbleBallToPoint` method defined in the `RobocupUtilities` class, passing the desired final position value as the parameter.

3.4.4 BShootBallToPoint Bean

The `BShootBallToPoint` bean is similar to the `BDribbleBallToPoint` bean except that the latter dribbles the ball, while the former kicks it with maximum force. The `BShootBallToPointBeanInfo` class provides similar options to the user as the `BDribbleBallToPointBeanInfo` class. This bean executes the `shootBallToPoint` method in the `RobocupUtilities` class.

4 Testing and Results

Using the developed components in `SoccerBeans`, we designed a basic soccer agent that was capable of dribbling a ball to its goal. The agent could also shoot the ball to a desired point in the soccer field if other players surrounded it. Figure 3 shows the agent design on the BDK.

The agent can perform three activities: `observe`, `dribble` and `shoot`. These are ordered from top to bottom in Figure 3. For every cycle in the agent's scheduling mechanism, if the player has received a new sensor input from the soccer server, then the BDI part of the mechanism selects the `observe` activity for execution. For action event inputs, the `dribble` and `shoot` activities are selected. The `dribble` activity is eligible for all action event inputs, while the `shoot` activity is eligible only for cases where other players are closing in on the agent. The subsumption part of the scheduling mechanism breaks the tie for such cases. The `shoot` activity always inhibits the `dribble` activity in the agent design.

The soccer game was played between two teams consisting of a single player, each designed as shown in the above figure. Successful experiments were conducted also for teams with a couple of players each, where the `shoot` activity was set to fire if any opponent approached the agent. Work is in progress for developing more components in `SoccerBeans` to enable the players to make non-trivial decisions for advanced communication and coordination.

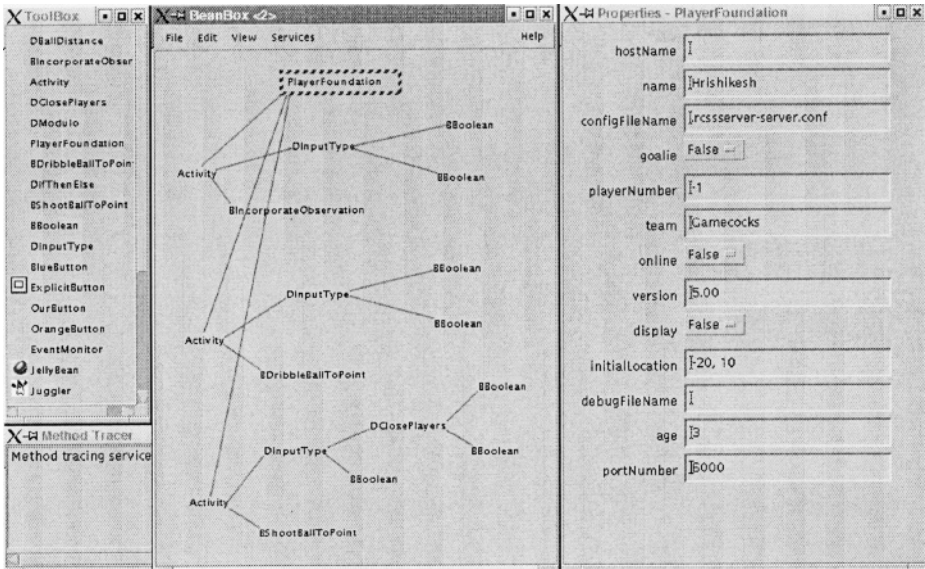


Fig. 3. Agent design using SoccerBeans. The red lines showing the connections between the components were added externally to the actual screenshot.

5 Related Work

Over the years, researchers have developed many agent frameworks for creating multi-agent systems. Some of the frameworks like JADE [2] and ZEUS [12] are domain-independent, but they restrict their users to designing agents and agent systems in a specific way. A user needs to learn the architecture of these tools before being productive, which is not a trivial task considering the complexity of these frameworks. The main objective of our work is to enable the user to design agent systems using CBAF-compliant frameworks with minimal learning. SoccerBeans builds on our previous work on Biter, an agent architecture for RoboCup. Biter implements a client for the RoboCup simulator, providing the basic functionality to design RoboCup teams. Some of its features include a world model with absolute coordinates, a graphical debugging tool, a set of utility functions, and a generic agent architecture supporting both reactive (subsumption) [4] and practical reasoning (BDI) [8] responses for scheduling activities. SoccerBeans inherits all of the above features from Biter, and extends the architecture by integrating it with a component-based architecture – JavaBeans. The University of Massachusetts’s JAF [10] project develops an agent framework using the JavaBeans technology. The JAF framework also provides components for designing disparate agents, but it is very restrictive in its methods for designing agents. In addition, the JAF system works only with the Multi Agent Survivability Simulator, a special test bed developed at the University of Massachusetts.

6 Conclusions and Future Work

We have introduced our CBAF specifications for developing agent frameworks, along with SoccerBeans—an implementation of CBAF for the RoboCup domain. The CBAF-compliant frameworks are unique in that there is virtually no learning curve for their users before they can start developing multiagent systems using the framework. In addition, the integration of the agent architecture with a component-based architecture like JavaBeans reduces the task of creating agents to that of visually linking the agents to its activities. These activities are also created by visually linking the necessary blocks of functionality from a pool of available components. Our system allows the user to quickly design and deploy his agents without worrying about the underlying architecture. CBAF seems ideal for research on applications such as trading-agent systems and resource allocation problems where there is a need for many agents with only slightly different functionality. We have demonstrated the ease and the usefulness of CBAF specifications with our SoccerBeans implementation for the simulated soccer application. We intend to complete the SoccerBeans framework in the future by developing additional components to add versatility to the available features for agent design. Finally, we view our system as a prototype for future agent-development environments that will merge the best techniques from agent-based software engineering and component-based design.

References

1. Biter: A robocup client.: <http://jmvidal.cse.sc.edu/biter/>.
2. Bellifemine, F., Poggi, A., and Rimassa, G.: Developing multi-agent systems with JADE. In: Proceedings of the Seventh International Workshop on Agent Theories, Architectures, and Languages, 2000.
3. Booch, G., Rumbaugh, J., and Jacobson, I.: The Unified Modeling Language User Guide. Addison-Wesley, 1999.
4. Brookes, R.A.: Intelligence without representation. *Artificial Intelligence*, 47:139–159, 1991.
5. Buhler, P., Vidal, J.M.: Biter: A Platform for the Teaching and Research of Multiagent Systems' Design using Robocup. In: RoboCup 2001: Robot Soccer World Cup V.LNCS/LNAI Lecture Notes Volume 2377. Springer Verlag, Berlin Heidelberg New York (2002).
6. Berners-Lee, T., Hendler, J., and Lasilla, O.: The Semantic Web. *Scientific American*, 2001.
7. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.
8. Georgeff, M., Pell, B., Pollack, M., Tambe, M., and Wooldridge, M.: The Belief-Desire-Intention model of agency. In: Proceedings of Agents, Theories, Architectures, and Languages, 1999.
9. Hendler, J.: Agents and the Semantic Web. *IEEE Intelligent Systems*, (16)2, 2001.
10. Horling, B.: A Reusable Component Architecture for Agent Construction. In: University of Massachusetts/Amherst CMPSCI Technical Report 1998-49. October, 1998.
11. JavaBeans: The Only Component Architecture for Java Technology. <http://java.sun.com/products/javabeans/>
12. Nwana, H., Ndumu, D., Lee, L., and Collins, J.: ZEUS: A tool-kit for building distributed multi-agent systems. *Applied Artificial Intelligence Journal*, 13(1): 129–186, 1999.

13. Soccer Server System.: <http://sserver.sourceforge.net/>
14. Vidal, J.M., Buhler, P.: Teaching Multiagent Systems using RoboCup and Biter. The Interactive Multimedia Electronic Journal of Computer-Enhanced Learning, (4)2, 2002.

Supporting FIPA Interoperability for Legacy Multi-agent Systems

Christos Georgousopoulos¹, Omer F. Rana¹, and Anthony Karageorgos²

¹ School of Computer Science, Cardiff University, P.O.Box 916,
Cardiff CF24 3XF, UK

{geolos,o.f.rana}@cs.cf.ac.uk

<http://www.cs.cf.ac.uk/Digital-Library/>

² Department of Computation, UMIST, Manchester, M60 1QD, UK
karageorgos@co.umist.ac.uk

Abstract. The conversion of a Multi-Agent System (MAS) into a FIPA-compliant system (i.e. one that adheres to FIPA standards) is important to support interoperability across different MAS. We provide an approach to undertaking such a conversion using FIPA-compliant gateways [7]. This approach avoids the need to re-write the entire legacy system to adhere to FIPA specifications. We propose an architecture for FIPA-compliant gateways that could be connected to a legacy MAS to provide automated FIPA interoperability. The use of the gateways is demonstrated within a Digital Library composed of multi-spectral images of the Earth, as part of the Synthetic Aperture Radar Atlas (SARA).

1 Introduction

The conversion of a Multi-Agent System (MAS) into a FIPA-compliant system (i.e. a system that adheres to FIPA standards) implies that system developers must rebuild their systems based on FIPA specifications. Such a conversion imposes amendments on the system architecture to conform to the new standards, which may result in extensive code rewriting and testing. This is often one of the limiting factors in promoting usage of FIPA standards.

We extend our previous work on the FIPA-compliant gateways [7], and describe an architecture of FIPA-compliant gateways that could be connected to a legacy MAS to provide *automated* FIPA interoperability. By the term *automated* it is meant that a developer would not need to have any knowledge of the FIPA specifications in order to make their system FIPA-compliant. For this purpose, a special *GatewayAgent* API [19] written in Java has been created to facilitate the realization of the FIPA-compliant gateways. Although, the proposed architecture of the generic FIPA-compliant gateways supports a limited number of performatives, a developer would be able to extend the gateway agent Java class in order to support any performative that it is not initially supported by the generic architecture.

Note that our approach should not be confused with agent software integration support for FIPA specifications, or with similar approaches that claim FIPA

compliance but actually alter[12] the original FIPA specifications. We briefly discuss the advantages and limitations of adopting our approach, and demonstrate how interoperability can be achieved with particular emphasis to the SARA (Synthetic Aperture Radar Atlas) system [15].

Although we are aware that standards are unlikely to remain static over time, FIPA provides the most valuable agent interoperability specification at the present time. Our approach is therefore focused on supporting interaction between agent systems that adhere to this standard.

2 Building FIPA-Compliant Gateways

Based on the guidelines provided by the FIPA association, for an agent platform implementation to be considered FIPA-compliant it must at least implement the “Agent Management” and “Agent Communication Language” specifications, which should conform to the latest *experimental* and/or *standard* status specifications.

The usual approach to conforming a MAS into a FIPA-compliant one is to modify the whole system based on FIPA specifications. A different approach that has so far been ignored is to adapt the architecture partially, and avoid an extensive re-write. Figure 1(a) represents a typical multi-agent system (MAS 1) that has been conformed to FIPA specifications in order to be able to interoperate i.e. receive/send data from/to other FIPA-compliant multi-agent systems (EXternal MAS). Figure 1(b) represents our approach to conforming a MAS into a FIPA-compliant one. The actual architecture of the system remains the same as before, but two FIPA-compliant gateways (in grey) have to be added to the system. These work as *adaptors (wrappers)* to ensure interoperability with other FIPA-compliant external multi-agent systems (EX MAS). Interoperability in this sense applies at both the communication and application levels. The communication level comprises the connection and communication layer, whereas the application level comprises the ontological and agent service layer [4].

The two *gateways* are the FIPA-compliant part of the system. Each of these has all of the mandatory, normative components of the FIPA architecture. The use of these FIPA-compliant gateways is illustrated in figure 5 of section 4.2 – where the adoption of the FIPA-compliant gateways in the SARA system is demonstrated. Each *gateway* contains three agents: the Agent Management System (AMS), the Directory Facilitator (DF) and the *gateway* agent. The AMS and DF are the FIPA agents, as defined by FIPA specifications. The *gateway* agent is the only agent of the system registered by both AMS and DF, and acts as a wrapper between MAS2 and any external MAS. All the available services of the system are represented by this agent. It is similar to a FIPA compliant system with only one registered agent capable of providing services. The Directory Facilitator (DF) and Agent Communication Channel (ACC), support the required infrastructure for enabling service interoperability, and are part of the

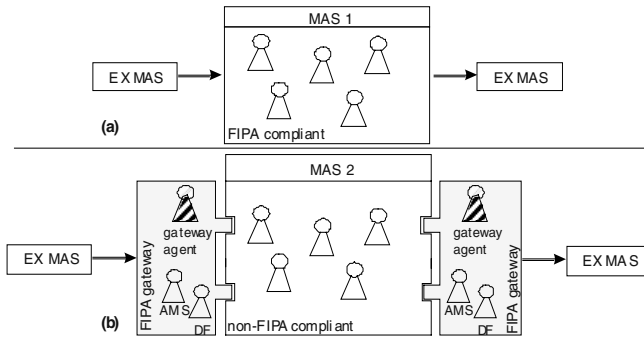


Fig. 1. Two different approaches of conforming an agent platform into a FIPA-compliant one

FIPA specifications. The communication between an EX MAS and MAS2 is accomplished through the Agent Communication Channel (ACC) and the protocols that are supported are reflected through the platform address. The *gateway agent* communicates with agents from EX MAS using the FIPA Agent Communication Language (ACL). Its responsibility is to translate the incoming messages to a form understood by its internal agents i.e. the agents that are hidden by the EX MAS. Likewise, the internal agents' requests have to be also converted by the *gateway agent* into ACL messages, in order to be understood by an EX MAS. The *gateway agent* maintains a list of the agents within the system being wrapped, along with the registered services (with DF) that each of them can provide. Therefore, based on the service requested by an EX MAS, the *gateway agent* knows to which system agent the message should be forwarded, after it has been translated into the form understood by the appropriate agent that receives the request.

Hence, the external MAS does not *see* anything else apart from the *gateway agent*; which on receiving a request from an external MAS (on the left side of MAS2) is responsible for transferring the request to the agents of its system, which are hidden by the external MAS, for processing the request. Once the request is accomplished, a response is returned to the external MAS through the *gateway agent*. In the case where agents from MAS 2 need to communicate with an external MAS (on the right side of MAS2), their request is passed through the *gateway agent* and translated into ACL; the results gathered by the external MAS are returned to MAS 2 agents through the *gateway agent* as well.

The *gateway agent* also supports agent conversation sessions by supplying the conversation ID (of its communication with the external agent) to its appropriate internal agent along with the translated message. Once, it receives feedback from one of its internal agents, it replies to the corresponding external agent on the conversation indicated by the conversation ID received by the former one i.e. the conversation ID that the *gateway agent* had initially sent to its internal agent.

2.1 Supporting Multiple Gateway Agents

Although one of the advantages of the FIPA-compliant gateway is to *isolate* the externally accessible part of the architecture i.e. the gateways, from the rest of the

system for increasing security (since the policy of the architecture remains hidden to a foreign Agency), some developers might need to *expose* more than one agents to an external MAS.

This could be achieved by adding multiple gateway agents to the FIPA-compliant gateway that provides interoperability between the legacy MAS and an external one, as shown in figure 2a. In this case, the agent that would need to be directly accessed by an external MAS, and could be represented by a separate gateway agent. For instance, with reference to figure 2, agent1 with service1 is resented by gateway agent1 (GA), service2 of agent2 by GA2 and service3/4 & 5 by GA3.

Even in the case where all of the available services provided by a legacy MAS are represented by a single gateway agent, the introduction of multiple gateway agents with replicated services in the FIPA-compliant gateway may also be useful for:

- Balancing the incoming requests among the existing gateway agents. In a MAS with numerous received requests, the gateway agent that receives a request from an EX MAS may pass the request to another (less occupied) gateway agent. For instance, the steps that have to be followed in order for a message to be passed from one gateway agent to another one, see figure 2b, are:

Step 1: An agent from an EX MAS sends a request to GA1.

Step 2: If the message is not understood by GA1, it replies to the sender agent with a

Not-understood message, otherwise it sends an Agree message including the

parameter *reply-to* with the gateway agent's name to which the message is

forwarded i.e. GA2. Therefore, subsequent messages (from the external agent)

will be directed to GA2.

Step 3: GA1 forwards the external agent's message to GA2 via an *Inform* message including the parameter *reply-to* with the external agent's name.

Step 4: GA2 communicates with its appropriate internal agent according to the service

required. The message that is sent to the internal agent is the content of the GA1's message, which has already been translated by GA1 (to validate the

external agent's message) to the form understood by their internal agents.

Step 5: GA2 upon receipt of results from its internal agent, generates an ACL message

and sends it to the external agent via an *Inform* message.

- Increasing fault tolerance of the interoperability part of a legacy MAS. The FIPA-compliant gateways may be configured to be distributed i.e. each gateway agent to be distributed on a different host. Therefore, even if one of the gateway agents fails, the MAS may still be able to provide its services to an external MAS through the rest of the gateway agents.

To conclude, there are three case scenarios for the FIPA-compliant gateway that provides interoperability between the legacy MAS and an external one: (a) a single gateway agent with all the available services registered under its entity (b) a gateway

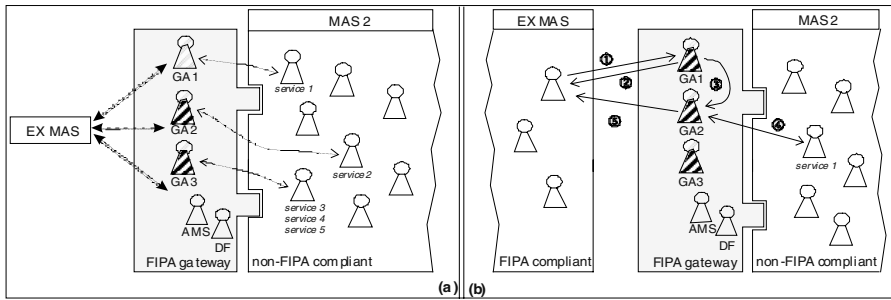


Fig. 2. Multiple gateway agents

agent per service (c) multiple gateway agents with replicated services. According to the MAS that need to address FIPA interoperability, developers can choose one of the above scenarios that suit their needs.

2.2 Advantages and Limitations of the FIPA-Compliant Gateways

The proposed approach of using the FIPA-compliant gateways for conforming a legacy MAS into a FIPA-compliant one, yields the following advantages:

- *Automatic FIPA interoperability with no or limited knowledge of FIPA specifications.* The adoption of the FIPA-compliant gateways automatically enables a legacy MAS to be FIPA compliant, capable of interoperating with any FIPA-compliant system. The inheritance of FIPA compliance by a legacy MAS involves the creation of the gateways and the configuration of corresponding gateway agents. The creation of a gateway is achieved by the execution of a simple script, where the configuration of a gateway agent involves a few code-lines. Therefore, a developer does not have to have any knowledge of the FIPA specifications for conforming a legacy MAS to a FIPA-compliant one. Consequently saving time in terms of reading, understanding, applying the FIPA specifications to the MAS that needs to address FIPA compliance and testing its interoperability. Limited knowledge of FIPA specifications will be required for extending the default gateway agent to support performatives currently not provided by the GatewayAgent (GA) API. This concerns knowledge on the ACL message structure and the performative's specifications that needs to be supported by the default gateway agent, as specified by FIPA.
- *System's architecture remains the same as before.* Implementation is only needed for the gateway agent and its interaction with the internal agents of the system that provide a service represented by the former. The FIPA-compliant gateways introduce FIPA compliance to a legacy MAS without influencing its original architecture. The interoperability part of the architecture (i.e. the gateways) is isolated from the rest architecture of the system. Based on FIPA, developers should conform to the latest specifications to guarantee a 100% FIPA-compliant system. The continuous improvement of FIPA specifications has a direct affect on the developer's systems since they should conform to the latest specifications. The

advantage of isolating the gateways from the rest of the system's architecture implies firstly that the original architecture of the corresponding legacy MAS is kept intact and secondly that any new standards (the FIPA revised specifications) which may be released in the future could be covered by a newer release of the GA API.

- *Security is increased.* There are still no coherent agent security details from FIPA at this time. Although, FIPA is planning in the future to investigate security related issues within FIPA architecture, and formulate a long term strategy for the integration of security features into FIPA specifications[3], [10], there is currently debate as to whether a generic or default level of agent security ought to be specified. It is also required that such security criteria be applicable to different types of agent infrastructures and application domains[13]. Based on the gateways approach, isolating the interoperable part of the architecture (i.e. the gateways) from the rest of the system increases security. The policy of the architecture remains hidden to the foreign Agency due to the FIPA-compliant gateways which act as a shield for the core system. The interaction between the system and a foreign agency is managed by the gateway agent; the rest of the agents, hardware/software resources cannot be accessed. Securing the FIPA-compliant gateways, from where foreign malicious agents can enter into the system, implies minimum security for the rest of the system. Imagine that agent authentication handled by the gateway agent for agent conversations/migrations could work as a firewall for the legacy MAS to restricts access on agents (instead of ports, as a traditional firewall does). The more secure the FIPA-compliant gateways are, the less security is needed for the rest of the system. For instance, the cost of encrypting the messages transmitted between the agents, apart from the gateway agent, can be avoided. Consequently, the minimization of security (apart from the FIPA-compliant gateways) also increases the overall performance of the system. Requirements and design issues for adding security to FIPA agent systems can be found in [13].

The limitation of the gateways' approach lies on the kind of systems that need to address FIPA interoperability, due to the limited performatives supported (7 out of 22) by the default GA API. Legacy systems that can automatically inherit FIPA compliance by adopting the FIPA-compliant gateways are agent-based systems that their need for interoperable communication with foreign FIPA-compliant systems is mainly based on the request of information. Those systems may be database archives managed by agent-based systems, Digital Libraries such as SARA active DL. Systems that require complex interoperable communication i.e. e-commerce or e-market which involve the negotiation, co-operation or co-ordination of heterogeneous agents can be supported by the gateways' approach, if and only if the default gateway agent is extended to support the performatives required.

3 Steps of Deployment

The deployment of the FIPA-compliant gateways involves the following steps: (a) the creation and configuration of the two FIPA-compliant gateways i.e. one to support interoperability between an external MAS and the legacy one, and vice-versa, and (b)

the creation of each of the gateway agents i.e. one per gateway. To facilitate the realisation of the FIPA-compliant gateways the *gateway_setup* script has been developed for the setup of the gateways and the GA API for the configuration and maintenance of the gateway agents.

3.1 Creation of the FIPA-Compliant Gateways

As mentioned in section 2, an agent platform implementation to be considered FIPA-compliant, it must at least adhere to the latest FIPA “Agent Management” and “Agent Communication Language” specifications. Therefore, the gateways should also adhere to those specifications. The creation of the gateways, that will adhere to those specifications, may be easily achieved by using a toolkit like FIPA-OS[5]. The *gateway_setup* script installs and configures the FIPA-OS toolkit on behalf of a developer. The developer has only to specify the directory on which FIPA-OS will be installed, a name for its platform and a list of the external platform-names that its MAS will need to interoperate with.

Once the configuration of the toolkit is finished, the execution of a simple FIPA-OS script starts-up the configured FIPA-agent platform (FIPA-compliant gateway) with the AMS and DF agents initialized. The last piece remaining for the implementation of the FIPA-compliant gateways are the gateway agents.

3.2 Creation of the Gateway Agent That Supports Interoperability between an EX MAS and the Legacy One

An example of a simple gateway agent written in Java using the GA API is demonstrated below. Actually, the following code example shows the implementation of the SARA gateway agent i.e. EXSA.

Example code of the SARA EXSA gateway agent

```

1  import GatewayAgent.*;
2  ...
3
4  public class EXSA
5  {
6
7      public void initialise()
8      {
9          GatewayAgent EXSA;
10         IEXSA_serv exsa_serv=null;
11
12         try // get a proxy for that class
13         {
14
15             exsa_serv=(IEXSA_serv)Namespace.lookup("//localhost:800/EXSA_serv");
16         }
17         catch(Exception e) {}
18
19         LinkedList properties=new LinkedList();
20         properties.add("EXSA");

```

```

20  properties.add("serve_EXMAS");
21  properties.add("EX_SARA_ontology.dtd");
22  properties.add(exsa_serv);
23  properties.add("EXSA_URA");
24
25      // Set-up the SARA EXSA Gateway agent
26  EXSA=new GatewayAgent("c:/fipaos/profiles/platform.profile"
    , "EXSA", "SARA");
27  EXSA.addProperty(properties);
28  ...
29  }
30  }

```

The commands necessary for the configuration and initialization of the gateway agent are in bold-italics. Firstly, the GatewayAgent library must be imported (line 1). In line 9, EXSA is declared as a gateway agent and is constructed in line 26 by calling the constructor of the GatewayAgent with the following parameters: the location of the *platform.profile* i.e the FIPA-OS configuration file which contains information about the FIPA-agent platform (gateway) installed, a unique name for the gateway agent and a name for its owner. Once the gateway agent has been created, it should be configured i.e. be informed of the available services provided by its internal agents. The *addProperty* method (line 27) of the GatewayAgent configures the EXSA agent based on the information provided by the properties LinkedList. Every LinkedList that is passed as a parameter to the *addProperty* method should hold information for a single service and its content should contain the following details in order:

- i) service-name ii) service-type iii) service's ontology iv) the internal agent that provides the corresponding service i.e. its proxy v) the internal agent's method that will be called once a request from an external MAS is received by the gateway agent.

as declared in lines 18-23, in this example, for the SARA gateway agent's service.

The gateway agent may be configured with more than one services; for each service the *addProperty* method should be called with input parameter a list of the particular service's detailed properties that need to be registered under the entity of gateway agent to its platform's DF. The gateway agent maintains a property list with content all of the registered services under its entity along with their private details e.g. the representative internal agent of the corresponding service, its ontology etc. The GA API supports dynamic service addition, deletion and updating. After the configuration of the gateway agent has been successful, it automatically registers itself with the AMS and the DF of its platform. Therefore, the steps of setting-up a gateway agent with the use of the GA API could be achieved within a few code-lines which involve its creation and configuration. The GA API provides multiple methods for its configuration and maintenance which can be found in [19].

At this point the gateway agent is automatically capable of handling the communication with an external FIPA-compliant MAS regarding a request or a cancellation of a prior request received from the later. This is due to the limited performatives supported by the default GA API. The following two sections describes how a default gateway agent (generated using the GA API) handles a request for a service, discusses the supported performatives and demonstrates how it is possible to extend a gateway agent Class for supporting other performatives that the ones supported initially by the GA API.

Performative Handling by the Gateway Agent. Once the gateway agent receives a request from an external FIPA agent, it locates its appropriate internal agent that can serve the specified request based on the *property* list the gateway agent maintains. The content of the received ACL message is parsed by the gateway agent against the ontology specified by the requested service and if it is valid, the gateway agent forwards it to its internal agent specified by the requested service's properties. After the request has been accomplished by the service's internal agent representative, the results are sent to the gateway agent. The gateway agent then generates an ACL message with content the results received by its internal agent and gives feedback to the external FIPA agent that initially placed the request. The interaction of the gateway agent with the external FIPA agent is handled by the REQUEST performative the GatewayAgent supports. A developer will only have to implement the method of the agent that represents the requested service indicated by the service's properties, in this case the EXSA_URA method (of the EXSA agent). The method should be of the form:

```
public String EXSA_URA(String do_undo,String message,String
                        convID)
```

This method receives as parameters the content of an ACL message subjects to the corresponding service's ontology, the conversation ID of the external FIPA agent with the gateway agent, and a String of value *do* or *undo*. The conversation ID may be used for supporting conversation sessions i.e. to identify whether a request is related with a prior one. The *do_undo* variable stands as a 'flag' which indicates whether a REQUEST or a CANCEL performative has been received, with values *do* or *undo* accordingly. Therefore, according to the *do_undo* value the method should either carry out (i.e. do) or cancel the task indicated by the *message* variable. Finally, the method should return a String containing the results of the task that has been carried out. Instead, a 'positive' or a 'negative' value should be return in the case where a task has to be canceled; the return value is determined based on the successful cancellation of the task.

Performatives Supported by a Default Gateway Agent. The generic FIPA-compliant gateways support a limited number of performatives. A default gateway agent created using the GA API is automatically enabled of handling a request or a cancellation of a prior request received from an external FIPA agent. This involves the support of seven out of the twenty two performatives currently provided by the standard FIPA Communicative Act Library Specification[18], namely: Agree, Cancel, Failure, Inform, Not-understood, Refuse, Request.

The interaction of a default gateway agent with an external FIPA agent over a request or cancellation of a service is handled by the REQUEST and CANCEL interaction protocols accordingly, as defined by FIPA specifications. Figures 3 and 4 show the FIPA interaction protocols as they have been implemented by the GA API. The figures show the flow of performatives exchanged between the default gateway agent and an external agent based on the events denoted in italics.

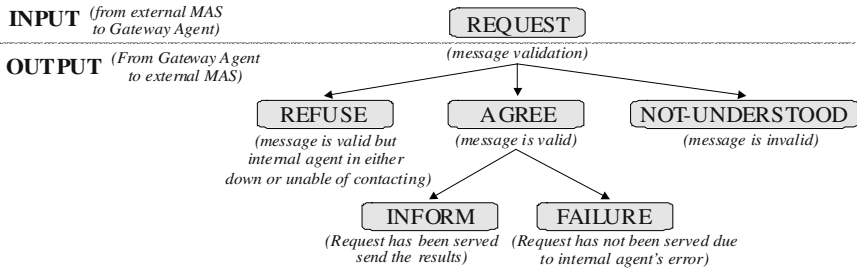


Fig. 3. FIPA Request interaction protocol

Of course this information does not concern a developer, since s/he does not have to have any knowledge of the FIPA specifications for conforming a legacy MAS to a FIPA compliant one, due to the FIPA-compliant gateways which are by themselves conformed to FIPA specifications.

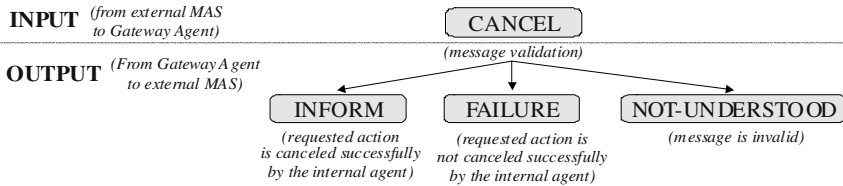


Fig. 4. FIPA Cancel interaction protocol

Though, a developer is capable of extending the default gateway agent to support other performatives than the ones currently provided by the initial GA API. Based on the GA API, the following Java Class template must be used for any performative that needs to be supported by the default gateway agent.

Java Class template of a performative

```

1 package GatewayAgent;
2
3 import fipaos.agent.conversation.*;
4 import fipaos.agent.task.*;
5 import java.util.*;
6
7 public class PERFORMATIVEperf extends Task
8 {
9     private Conversation conv;
10    LinkedList properties;
11
12    public PERFORMATIVEperf (Conversation conv, LinkedList properties)
13    {
14        this.properties=properties;
15        this.conv=conv;
16    }
17
18    protected void startTask()
19    {

```

```

20         // developer's code here
21     }
22 }

```

The Class must have the name of the performative that needs to be supported accompanied by the ‘*perf*’ string. For instance, to support the PROPOSE performative the Class-name must be *PROPOSEperf*. The code that will be executed upon the receipt of the particular performative by the gateway agent should be placed inside the *startTask* method, in line 20. Finally, the gateway agent must be informed of the new supported performative. This is done by calling the *setPerformative* method of the GA API e.g. *gateway_agent.setPerformative(PROPOSE)*. Consequently, once the gateway agent receives a message from an external FIPA agent containing the *PROPOSE* performative, the *PROPOSEperf* Class will be initiated. The gateway agent replies to an external FIPA agent with an NOT-UNDERSTOOD performative when it receives a message of an unsupported performative by itself.

The Class template provides the *conv* and *properties* variables (initialised in lines 14 and 15). The *conv* variable contains information regarding a message received from an external FIPA agent like the sender agent-name, the content of the message e.t.c., where the *properties* variable contains the gateway agent’s *property* list (mentioned in 3.2 section). The *GAParse* method of the GA API may be used to validate the incoming message (included in the *conv* variable), where the FIPA-OS API may be used for structuring and sending a reply ACL message to the external FIPA sender agent.

Extending the default gateway agent requires knowledge on the ACL message structure and the performative’s specifications that needs to be supported by the gateway agent, as specified by FIPA.

3.3 Creation of the Gateway Agent That Supports Interoperability between the Legacy MAS and an External One

This gateway agent does not provide any services. Its responsibility is to use services provided by external FIPA compliant MAS on behalf of the agents of the legacy MAS. The implementation of the gateway agent involves its creation (lines 1,9,26 of code example in 3.2 section) and configuration.

The method of the GA API that configures a gateway agent is the *setEXservices*, see [19]. This method receives as parameters information regarding the services provided by an external FIPA MAS which are intended to be required by the agent of the legacy MAS i.e. a list of services’ names, the Directory Facilitator’s name and a list of the communication protocols supported by the external FIPA MAS that provides the specified services. For service(s) provided by another external FIPA MAS, the *setEXservices* method must be called again with input parameters the information of the corresponding MAS. The gateway agent maintains a list of all the external services along with their detailed information configured under its entity.

When an agent from the legacy MAS needs to use an external service, its request is passed to the gateway agent by calling the *sendRequest* method of the GA API. The gateway agent then is responsible of making contact and handling the communicating with the external FIPA agent that provides the particular service specified by its

internal agent to accomplish the request. The communication protocol over which the communication of the gateway agent with the corresponding external FIPA agent is based is defined in the gateway agent's configuration details, where the name of the external FIPA agent that represents the requested service is traced by the gateway agent. This is done by interrogating the DF of the external MAS of which agent of its system handles the particular service. The *sendRequest* method returns the results of the service requested by an internal agent, where a *not-understood* or a *failed* message-string is returned if the internal agent's request has not been understood or failed to be accomplished by the external FIPA agent.

Note that an internal agent's request before it is forwarded to an external FIPA agent has to be translated to the form understood by the later based on its service's ontology. As well as the results received from a service provided by an external FIPA agent have to be translated into the form understood by the agents of the legacy MAS. Since the ontology of a service is service-dependent this translation is impossible to be made by the gateway agent itself. For this purpose the developer must create a Class that will enable the translation process. Consequently, any message that is passed on the *sendRequest* method or extracted from the content of an ACL message (holding the results of a requested service) have to be first parsed by the developer's Class so as to be understood by either of the agents i.e. an external FIPA agent or an agent from the legacy MAS. For instance, when the *sendRequest* gateway agent's method is instantiated, it sends a REQUEST to the external FIPA agent that provides the service indicated by *ex_service_name* parameter with content as the content of the *message* parameter; which must have already been translated by the developer's Class.

4 Testing the Interoperability of the FIPA-Compliant Gateways

Our research is based on the Synthetic Aperture Radar Atlas (SARA) active Digital Library[14], [15]. In order to achieve interoperability between our system and an external one, we have adopted the generic FIPA-compliant gateways approach. In the following section we give a brief discussion of SARA project, we demonstrate how interoperability can be achieved by using the approach outlined previously and we present results of experiment tests performed on the interoperability of our system.

4.1 The SARA Active Digital Library

SARA is an active digital library of multi-spectral remote sensing images of the earth from the SIR-C Shuttle mission, which provides web-based online access to a library of data objects at Caltech, the San Diego Supercomputer Center, and the University of Lecce in Italy. The objective of the SARA project is to develop an infrastructure for a high-speed, high-volume, multi-protocol distributed database, together with a means to attach distributed computing resources for data conversion, visualization and knowledge discovery[17].

A prototype MAS, which comprises both intelligent and mobile agents, has been developed to manage and analyse distributed multi-agency remote sensing data; more information can be found on our web-site[9]. The SARA architecture (figure 5) is

composed of a collection of *information* and *web* servers, each of them having a group of agents, Local Interface Agents (LIA) and User Interface Agents (UIA) accordingly.

We separate mobile agents from stationary service agents. Our approach is to localize the most complex functionality in non-mobile LIAs, which remain at one location, providing resources and facilities to lightweight mobile agents that require less processor time to be serialized and therefore quicker to transmit. LIAs are stationary agents that provide an extensible set of services and a level of abstraction between resource servers and requesting mobile agents. UIAs provide a front end to the end user, for checking the user input and displaying the results.

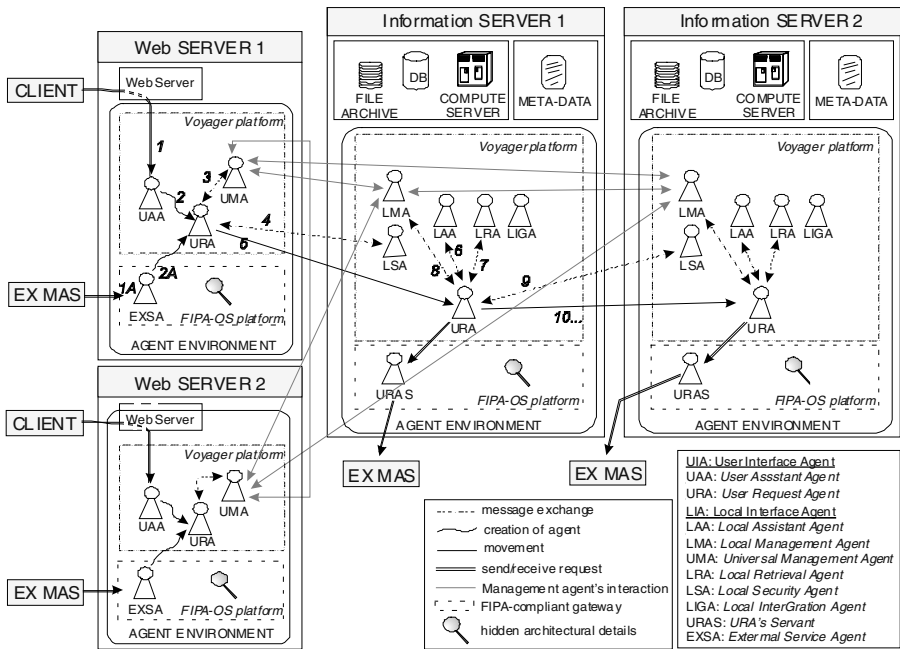


Fig. 5. The FIPA interoperable SARA architecture

4.2 SARA and FIPA Compliance

The introduction of FIPA interoperability into the SARA system enables it to communicate with other MAS and vice-versa. The union of SARA system with other MAS extends its capabilities by providing users with further information. For instance, information retrieved from the SARA system can be further enhanced by additional information gathered from a GIS system that is capable of interoperating with SARA. The longitude and latitude of a particular area of the earth can be used as parameters on a GIS (Geographic Information System) to retrieve land information such as street names, which can then be combined with the image based on

geographical coordinates in SARA, resulting in a detailed map of the particular area. Likewise, an external MAS can interoperate with SARA and use its information.

The interoperability of the SARA system is based on the use of FIPA-compliant gateways which are implemented using the GA API. The architecture of the SARA system with added FIPA interoperability is depicted in figure 5. An external multi-agent system (EX MAS) can interoperate with SARA through the FIPA-compliant gateway (outlined by the dashed box) which is placed on every Web-server, where SARA can interoperate with an EX MAS through the FIPA-compliant gateway which is placed on every Information-server. The architecture of the FIPA-compliant gateways which is a slight variation of the architecture of FIPA-OS *configuration case 2*[6] is depicted in detail in figures 6; since the FIPA-OS toolkit has been used as the FIPA agent platform for the realization of the FIPA-compliant gateways.

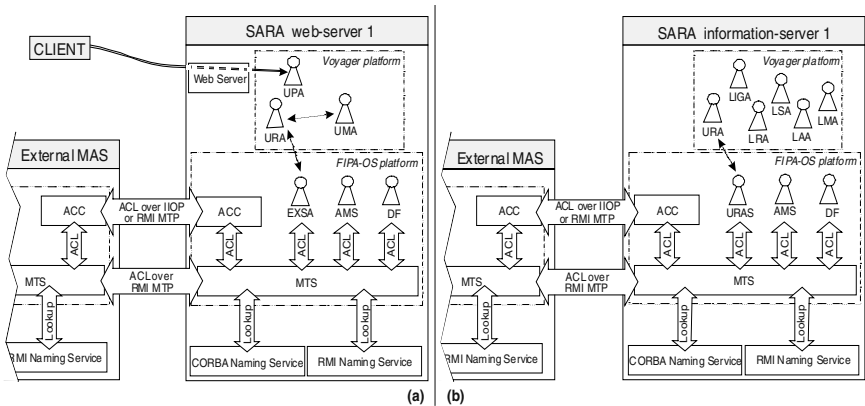


Fig. 6. Representation of the FIPA-compliant gateways: (a) on the Web-server and (b) on the Information-server

The EXSA agent is the gateway agent of the FIPA-compliant gateway placed on every Web-server. The service provided by EXSA is the retrieval of a collection of Earth images from the SARA DL based on specific coordinates. EXSA can be considered similar to UAA; based on the fact that, as a user is represented by a UAA, an external MAS is represented by an EXSA. Once, EXSA receives a valid request from an external FIPA agent it creates a URA and forwards its request to the later (after it has been translated by the EXSA to the form understood by URA). Then, URA works in the same manner as if it would have been created by UAA i.e. by starting its itinerary and migrating through the information servers for accomplishing its task. When URA finishes its job, it sends the results back to the EXSA. Finally, EXSA constructs an INFORM ACL message with content the information gathered by URA and replies to the external agent from where the request has been initially placed. The resource access level and request’s priority level is according to the EX MAS that accesses SARA.

Finally, the URAS agent is the gateway agent of the FIPA-compliant gateway placed on every Information-server. The purpose of this agent is to server URA with information gathered from external FIPA-compliant MAS. When URA needs to access an EX MAS, its request has to be first translated to the from understood by the

external FIPA agent based on the later service's ontology, before it is send to URAS. Once, URAS receives a request from a URA it comes in contact with the appropriate external FIPA agent to accomplish URA's request. By the time URAS has not acquired the results requested by URA, URA is free to continue with its next task (if it has one), migrate to another Information-server or wait for URAS agent's response. The list of the external multi-agent systems that SARA can interoperate with is controlled by the SARA management agents. Therefore, URA has the right to come in contact with URAS if and only if one or more external MAS is listed in its itinerary.

4.3 Experiment Test Results

To test the interoperability of SARA system with an external one we have conducted our experiments using two different types of agent FIPA-compliant platforms. The first one was implemented using FIPA-OS toolkit (version 2_1_0-20030219000011, build:314) running on Unix and the second one was implemented on JADE toolkit[11] (version 2.4.1) running on Linux.

The tester agent of the FIPA-OS agent platform was a simple agent that was created to search the DF of SARA system for the EXSA's service and perform a Request. The second tester agent was developed using the JADE agent building toolkit and located at the Manchester Agentcities[1] node, which is hosted at the Dept. of Computation, UMIST (University of Manchester Institute of Science and Technology) [2].

The screenshots in figure 7 show the results of our experiments. The top picture is the console server of the SARA web server (running on Windows XP), the middle one is the console of the SARA information server (running on Unix) and the last one shows the execution of the tester agent of the FIPA-OS agent platform (running on Unix).

Initially, both of the tester agents performed a search on the DF of SARA to find the EXSA gateway agent's AID (Agent Identifier) that provides the appropriate service. The interaction of an agent with the SARA DF is managed by FIPA-OS itself. Once, the tester agents have acquired the gateway agent's AID, they both sent a *Request* performative to EXSA, similar to the following one:

Example of a simple Request ACL message

```
(request
:sender agent_from_EX MAS_id
:receiver EXSA_id
:content (<?xml version="1.0" ?><ex_SARA_mes>
          <coordinates c1="16.317" c2="107.654" c3="16.061"
                    c4="108.082" c5="16.828" c6="108.575"
                    c7="17.087"
                    c8="108.144" /></ex_SARA_mes>)
:language XML
:ontology EX_SARA_ontology.dtd
...)
```

The coordinates specified in the content of the ACL message corresponds to the collection of images required by the sender agent. When EXSA received the requests from the tester agents (figure 7a) it validated them. Since the incoming requests were valid, it replied to each of the tester agents with an *Agree* performative (figure 7c) and created for each of them a proxy of the URA agent locally. URA is the *internal* SARA agent that accepts as input Earth coordinates and gives as output a collection of images corresponding to the coordinates provided. The messages sent to each URA from EXSA were: the tester agents' request translated into the form understood by URA (the XML content of the message) and the conversation ID of the corresponding tester agent's interaction with EXSA.

After each URA has been initialized by EXSA, it communicated with its local management agent i.e. UMA, in order to receive the itinerary that has to follow through the SARA information servers in order to accomplish its task i.e. gather the information requested by EXSA. UMA is responsible for constructing every URA's itinerary in SARA according to the information provided by the latter and the current status of the system (known by UMA), i.e. availability of resources, server failures, number of agents on each server. UMA may also direct URA to collect the results of its query from a server which have already been stored by a previous agent having a similar query. The management agent's (UMA, LMA) interaction is described in [8].

For each URA, the steps of accomplishing their task may be traced by following the numbers on the diagram of the SARA architecture in figure 5 or with reference to the SARA information server's console in figure 7b. The console records the execution of every URA agent on the visited SARA information servers and reveals their interaction with the local stationary agents hosted by Voyager[16] agent platform.

After URA has accomplished its task, it sent the results back to the EXSA along with the conversation ID, initially received by EXSA, and self-terminated. Then EXSA replied to each of the tester agents based on the conversation indicated by the conversation ID received from its internal agent i.e. URA, via an *Inform* performative including a URL address (see figure 7b and 7c). The actual results could be acquired by accessing the corresponding URL address.

Details on the messages exchanged between the tester agents and the EXSA gateway agent, the translation of a *Request* ACL message performed by EXSA to the form understood by URA i.e. XML, and an example of results gathered by URA based on specific coordinates can be found in [19].

5 Conclusion

In this paper we described how a developer may adopt the generic FIPA-compliant gateways approach for achieving automated FIPA-compliance to a legacy MAS. We discussed the advantages and limitations of the proposed approach and we demonstrated the successful interoperability provided by conducting experiment test on a MAS utilizing the FIPA-compliant gateways. Our future concern is to further extend the "gateways" by defining extra sets of operations that may be supported by these agents. For instance, the utilisation of a security layer will enable heterogeneous MAS to interoperate using X509 based digital certificates. In addition, an agent

mobility layer would provide the capability to support agent migration between heterogeneous MAS built on the same agent platform.

(a)

```

C:\project\WEB-INF\servlets>java WServer

Voyager Server is running..
Proxy for the EXSA agent has been created.
The EXSA agent (FIPA-compliant gateway) will be initialised.
31/03/03 07:17:52 EXSA: Now, I am initialised!
31/03/03 07:18:25 EXSA: A Request has been received.
-----
Agent details: (Agent-identifier :name EXMAS@bloodstone.cs.cf.ac.uk :addresses (sequence fipaos-rmi://bloodstone.cs.cf.ac.uk:3000/EXMAS ) >
Conversation ID: EXMAS@bloodstone.cs.cf.ac.uk:10491345732264
31/03/03 07:18:25 EXSA: Message with conversation ID:EXMAS@bloodstone.cs.cf.ac.uk:10491345732264 is understood.Trying to communicate with URA,and take the results.....
31/03/03 07:18:33 EXSA: Results have been transferred to EXMAS@bloodstone.cs.cf.ac.uk agent.
-----
01/04/03 04:00:29 EXSA: A Request has been received.
-----
Agent details: (Agent-identifier :name DFTester@Halkidiki.agentscities.org :addresses (sequence http://halkidiki2.co.unist.ac.uk:7777/acc ) >
Conversation ID: EXSA@gallium.cs.cf.ac.uk:492092126114
01/04/03 04:00:29 EXSA: Message with conversation ID:EXSA@gallium.cs.cf.ac.uk:492092126114 is understood.Trying to communicate with URA,and take the results.....
01/04/03 04:00:31 EXSA: Results have been transferred to DFTester@Halkidiki.agentscities.org agent.

```

(b)

```

Terminal
Window Edit Options Help
scmctl-% java Server
31/03/03 07:18:20 The Voyager server launched successfully.
31/03/03 07:18:20 The LAA's resource-check is enabled.

31/03/03 07:18:28 URA: (with id:gallium_8000_EXSA_1049134834217) trying to contact LAA & LRA...
31/03/03 07:18:29 LAA_con: generating JDBC connection, instructed by URA (with id:gallium_8000_EXSA_1049134834217)
31/03/03 07:18:30 Lra_EXquery: executing SQL query received by URA (with id:gallium_8000_EXSA_1049134834217)
31/03/03 07:18:31 Laa_discon: closing JDBC connection, instructed by URA (with id:gallium_8000_EXSA_1049134834217)
31/03/03 07:18:32 URA: Task accomplished. Sending the results to the appropriate UPA/EXSA agent...
31/03/03 07:18:32 URA: self terminating...

01/04/03 04:00:30 URA: (with id:gallium_8000_EXSA_1049209407389) trying to contact LAA & LRA...
01/04/03 04:00:30 LAA_con: generating JDBC connection, instructed by URA (with id:gallium_8000_EXSA_1049209407389)
01/04/03 04:00:30 Lra_EXquery: executing SQL query received by URA (with id:gallium_8000_EXSA_1049209407389)
01/04/03 04:00:30 Laa_discon: closing JDBC connection, instructed by URA (with id:gallium_8000_EXSA_1049209407389)
01/04/03 04:00:30 URA: Task accomplished. Sending the results to the appropriate UPA/EXSA agent...
01/04/03 04:00:30 URA: self terminating...

```

(c)

```

Terminal
Window Edit Options Help
scmctl-% java Agent_EXMAS /home/scmctl/fipaos/profiles/platform.profile EXMAS exmas
31/03/03 07:18:23 Searching SARA DF for EXSA service...
31/03/03 07:18:23 Service has been found.
31/03/03 07:18:24 Sending a Request to EXSA agent...
31/03/03 07:18:25 An Agree message is received.
31/03/03 07:18:33 An Inform message is received.

The results retrieved from EXSA agent:
http://www.cs.cf.ac.uk/user/C.Georgousopolost/gallium/EXSA/310303/gallium_8000_EXSA_1049134834217.xml

```

Fig. 6. Test results

References

1. AgentCities - a global, collaborative effort to construct an open network of on-line systems hosting diverse agent based services, <http://www.agentcities.org> (2003)
2. AgentCities node hosted by UMIST (University of Manchester Institute of Science and Technology), UK, <http://www.agentcities.co.umist.ac.uk> (2003)
3. Burg , B., Dale, J., Willmott, S.: Open Standards and Open Sources for Agent-Based Systems, Article in: Agentlink, news 6 (2001)
4. Charlton, P., Bonnefoy, D., Lhuillier, N., Gouaich, A., Camenen, Y.: Dealing with interoperability for Agent Based Services, White paper, <http://leap.crm-paris.com/agentcities/Resources/resou rces.html> (2000)
5. FIPA-OS, <http://www.nortelnetworks.com/products/announcements/fipa/index.html>
6. FIPA-OS Inter-platform Communications Configuration Guide, <http://www.nortelnetworks.com/ products/announcements/fipa/index.html> (2002)
7. Georgousopoulos, C., Rana, O. F.: An approach to conforming a MAS to a FIPA-compliant system. In First International Joint Conference on Autonomous Agents and Multi-Agent Systems - AAMAS 2002, ACM ISBN 1-58113-480-0, Italy, Bologna (2002) 968–975
8. Georgousopoulos, C., Rana, O. F.: Combining State and Model-based Approaches for Mobile Agent Load Balancing. In SAC 2003 - ACM Symposium on Applied Computing, ACM ISBN 1-58113-624-2, Melbourne, Florida, USA (2003) 878–885
9. <http://www.cs.cf.ac.uk/Digital-Library/>
10. <http://www.fipa.org/docs/output/f-out-00065/>
11. JADE (Java Agent DEvelopment Framework), <http://sharon.csel.it/projects/jade> (2003)
12. Panti, M., Penserini, L., Spalazzi, L., Valenti, S.: A FIPA compliant agent platform for federated information systems. In ACIS, volume 1, issue 3. Special issue on software engineering applied to networking & parallel/distributed computing, ISSN:1525-9293, USA (2000) 145–156
13. Poslad, S., Calisti, M.: Towards improved trust and security in FIPA agent platforms. Proceedings of Autonomous Agents 2000 Workshop on Deception, Fraud and Trust in Agent Societies, Spain (2000)
14. Yang, Y., Rana, O. F., Georgousopoulos, C., Walker, D. W., Williams, R., Aloisio, G.: Agent based data management in digital libraries. In Parallel Computing Journal, Elsevier Science, vol. 28, issue 5 (2002)
15. Yang, Y., Rana, O. F., Walker, D. W., Williams, R., Aloisio, G.: Towards an XML and Agent-Based Framework for the Distributed Management of Multi-Spectral Data. 6th International Digital Media Symposium, Bradford, UK (2001)
16. Voyager, Recursion Software, Inc., <http://www.recursionsw.com/osi.asp> (2003)
17. Williams, R.D., Sears, B.: A High-Performance Active Digital Library, Parallel Computing, Special issue on Metacomputing (1998)
18. FIPA Communicative Act Library Specification, <http://www.fipa.org/specs/fipa00037/>
19. http://www.cs.cf.ac.uk/Digital-Library/API_MessExchange.doc

Dynamic Multi-agent Architecture Using Conversational Role Delegation

Denis Jouvin and Salima Hassas

Laboratoire d'InfoRmatique en Images et Systèmes
d'information (LIRIS, formerly LISI),
Université Claude Bernard Lyon I,
43 bd du 11 novembre 1918,
69621 Villeurbanne, France
{djouvin,hassas}@lisi.univ-lyon1.fr

Abstract. This paper discusses the notions of dynamic composition and dynamic architectures, in the context of conversational multi-agents systems, as well as distributed component oriented or object based systems. The directory service or facilitator agent paradigm, commonly used for building architectures exhibiting these properties, is examined and discussed. It is then compared with a proposed alternative paradigm, based on dynamic conversational role delegation. It is shown that the directory service paradigm, among other weaknesses, exposes the system to synchronization problems when complex protocols are used or concurrent access to the directory are involved, and is not transparent. The role delegation paradigm, on the other hand, presents significant advantages, including a better synchronization with ongoing conversations, and allows transparent encapsulation of the compositional behavior. A working prototype, focused on electronic auction and on-the-fly protocol adaptation, through adaptation proxies, is presented to demonstrate the feasibility of the approach.

1 Introduction

In modern complex, open and distributed systems, it is now clear that *dynamicity* has become a very important requirement. Intuitively, we could define dynamicity in our context as the ability for a system to be configured, developed, maintained, modified *at runtime*, without compromising its integrity and ongoing processes. Obviously this also influences the way we define other important system properties, such as composability, and the notion of software architecture itself.

The reasons for such an interest in the dynamicity of system composition are many: modern systems are often too heterogeneous to allow a synchronized and uniform maintenance, or initial development. Components are maintained and integrated by different sources at different times. Some large systems, or even not so large systems such as PC operating systems, would imply a too high cost in time and annoyance, if they had to be recompiled or even shut down every time a slight change has to be

applied. Sometimes, the availability of a system is so critical that having to shut it down even for a short time is unacceptable.

To address this need, during the past few years, various techniques have emerged in the software industry: software plug-ins infrastructures, auto-upgrade features, and dynamically pluggable component based systems. These techniques are now found in many different application fields, such as operating systems, web browsers, media players, to name a few, not only in very complex and distributed enterprise systems. Mobile pieces of code, such as Java applets, servlets, Enterprise JavaBeans, and other component models like ActiveX/DCOM or .NET infrastructures, allow for an easier and automated deployment of components on the client side, as well as on the server side, and thus serve the same purpose: *making system modification more dynamic*.

In the research literature, system dynamic reconfiguration has been explored mainly within specific distributed component systems (see for example Conic [3]), where component composition is achieved through explicit point-to-point connectors.

In this paper, we start from the hypothesis that maintaining or modifying a system dynamically is best achieved by compositing, substituting or aggregating components. All the approaches mentioned above share the concern of *dynamicity*; however here we will mainly focus on Distributed Object Systems (DOS) and Component Oriented Systems (DCOS), and Multi-Agents Systems (MAS), which probably represent the most significant contributions on this matter. The underlying idea is to explore the notion of dynamic composition, and identify the underlying concepts and mechanisms worth standardizing in conversational MAS development. For this purpose, we propose to examine a powerful paradigm: *conversational role delegation*.

The paper is organized as follows: section 2 positions the notion of dynamic composition by comparing DOS/DCOS and MAS approaches, and then reformulates the problems and issues. Section 3 introduces the concept of dynamic architecture, by giving an overview of a common existing design and implementation technique, the component directory or facilitator agent paradigm, and proposes an alternative approach based on conversational role delegation. Section 4 presents our prototype implementation. We then conclude on the theoretical implications on interaction protocols, focusing on the need for protocol composition, supported by an adequate underlying interaction protocol theory and formalism.

2 Dynamicity and Composability in Existing Models

2.1 Component Oriented and Agent Oriented Approaches Complementarity

By briefly looking at the history of both of these approaches, we notice that they are quite complementary of each other regarding composability and dynamicity. This complementarity partially justifies the current trend of these communities to get closer to each other, since they share quite similar issues and objectives. The reader can refer to [9] for a more detailed comparison.

DOS and DCOS lack of Support for Synchronization. Distributed Object Systems inherit composability from Object Oriented programming. Object encapsulation provides a simple, yet efficient paradigm to decompose a system recursively into a graph of objects. The lack of constraints on this model makes it very natural to decompose a part of the system, represented by an object, into several subcomponents, implemented by other objects. However this model does not help managing temporal and synchronization complexity. Alternative composition constructs such as inner classes or interface delegation increase composability, but are not fully dynamic.

Distributed Component Oriented systems provide plugging and self-presentation capabilities to components, which is an important step toward system dynamic composition. Components are dynamically “plugged” into container components, and may adapt to this context and find other components on demand. Still, the component models do not provide a standard way to cope with synchronization problems, in case of components involved in ongoing communications. Most of the time, the client-server, or simple stateless service paradigm is assumed.

The fact is that real distributed systems do involve complex state-full components, asynchrony problems, as well as transactional communications. This is where modifying a system dynamically, replacing or aggregating a component, becomes difficult, and where the “flat component” models do not suffice. Dynamically reconfigurable distributed systems address this issue either by attempting to restore the component states, or by waiting for the component to be in a quiescent state where no transaction or state-full communication is interrupted.

Modern distributed object-based systems also include dynamic service discovery, and other related complementary services, in their infrastructures. These services allow some sort of dynamic late-binding and, thus, easier system architecture dynamic modifications. Such architectures are discussed in section 3. However, as with most of component oriented approaches, the synchronization between services represented by remote objects is not considered in the model.

Multi-agents Systems lack of (recursive) Composability. Composability is seen here as a stronger requirement than mere component interconnection: it is the ability for a component to be composed of several components transparently, *i.e.* a system of components to be perceived as *one* component (*composite atomicity*).

In Multi-Agents Systems, *dynamicity* is inherent to the approach. Thanks to agent autonomy and the way agents interact, this property is theoretically verified by the very nature of a MAS. Yet, the problem arises from the other side: most of MAS are not composable in a well defined way, at the agent level. They theoretically involve higher level organization and social interactions, from where compositional structures are supposed to emerge, which probably explains why this issue is still open [8]. Systems validating this property are often referred to as holonic MAS.

To address the code reuse necessity, MAS designers have used existing componential techniques extensively, to bring reusability to the intra-agent development level. This was strongly needed since multi-agent theories and models usually focus on agent interaction and organization rather than internal agent implementation considerations. The problem is that both technologies have been combined without really considering the overlapping part (functionally speaking). Components and agents have actually

lots in common, and the border between them is rather thin; in other words, it is not always easy to decide whether a given function or role should be implemented as a component inside an agent, or as a separate agent.

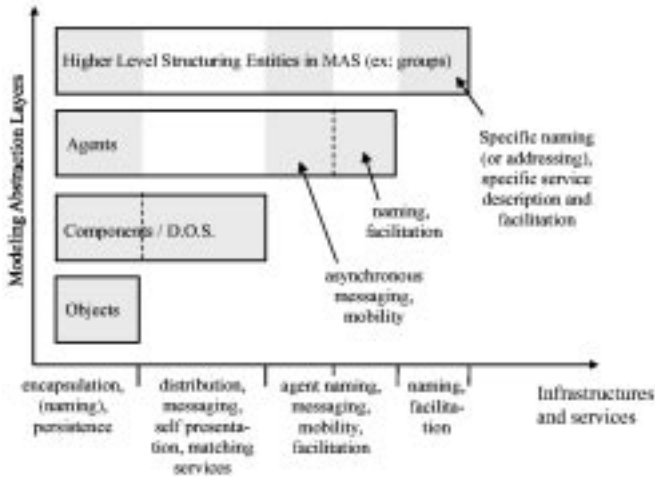


Fig. 1. Functional Redundancies between Component, Agent and Agent Group Layers

In modern MAS platforms, which combine componential approaches with MAS standards such as the FIPA architecture [5], several functions are indeed redundantly implemented at the component and agent levels, as figure 1 illustrates. Among them:

- asynchronous messaging, necessary in components managing conversation follow-up and protocol validation, and of course present at the agent level;
- component service description, self-presentation, matching or directory service, which are highly redundant with facilitator agents;
- component life-cycle management (naming, persistence, possibly mobility), which are redundant with agent life-cycle management

Table 1. Approaches Comparison Summary. The following table summarizes the characteristics of each approach regarding composability, scalability and dynamicity

	Distributed Object Systems / Component Based Systems	Multi-Agents Systems
Initial focus	(static) composability, scalability	dynamicity (inherent)
Recent evolutions	dynamicity; service matching / directory services.	behavior components libraries
Weaknesses	synchronization management during compositional change; design-time, not run-time	fixed design layers (component, agent, group); weak scalability; no composability at agent level



2.2 MAS Composability Increases MAS Scalability

Section 2.1 discusses the redundancy, but also, while not specifically naming it, the scalability problem. In our context, we see scalability as the ability for a system, or its components, to be easily complexified, by adding more components and functions, while preserving the initial design. To answer these problems, our propositions are:

- First, to unify and clearly partition the redundant functions between the component and agent layers. This is a standardization issue, and is out of scope of this paper;
- Second, to define standard composition mechanism(s) for agents, that would allow system designers to abstract the architecture from the design layer, and a smoother transitions from a component-only based subsystem to an “agentification” of that sub-system, or vice versa. This would increase the scalability.

To differentiate between the component and agent levels, our proposition is thus to consider that components are designed to be “plugged” at design time, less dynamically, whereas at the agent level, agents are always dynamically linked, allowing dynamic composition, reconfiguration, and higher level organization. This has two implications on the design and implementation of MAS:

1. Any component that is dynamically linked, through directory services or another mechanism, should be implemented as an agent; in other words, we state that this function (dynamic composition) in a MAS should belong to the agent level. This seems to us natural since a component involving dynamic composition will usually have all the characteristics of an agent;
2. Multi-Agents Systems, even when involving a complex organization and dynamic linking (as we will discuss in next section), should have their architecture or configuration described in some way, so that designers may understand the configuration *at a given time*, and may reverse back to a more static implementation easily.

The latter implication seems to us an open issue, since describing a highly volatile configuration is a difficult task. The former reflects the philosophy of our approach: whenever a component reaches a given complexity level, and present all the characteristics of an agent, it should be implemented as an agent.

A typical example are the conversation managers and conversation follow-up components that we usually find in MAS platforms — like FIPA-OS [11] *tasks*, Zeus [14] finite state machines, or Jade [7] *behaviors*. When the interaction protocol is not trivial, managing such conversation asynchronously in a reusable manner is so complex that the composition code becomes more complex than the agent code itself.

2.3 Refined Statement of the Dynamic Composability Problem

If we can define a composition mechanism at the agent level, then a complex agent, possibly made of complex components, would be easily replaced by a group of agents. The conditions for such a dynamic composition paradigm to be consistent with our previous requirements are:

1. To allow *seamless* composition, so that one does not need to change the rest of the system when an agent is substituted with a composite. This is achievable by allowing a group of agents to be viewed as a single (holonic) agent;

2. To achieve such composition in a *synchronized* manner, with respect to ongoing interactions, like transactions, conversations, etc.;
3. To keep agent *situatedness* property intact.

From these necessary conditions, we see that composition is closely related to the ongoing interactions of agents, and cannot be defined independently of that aspect.

Agent *situatedness* means here that an agent is partly defined by its situation with respect to its environment, in our case the conversational environment. A compositional relationship between agents or components should not be dissociated from the context in which it makes sense. Thus, this relationship should not be global, but viewable and usable only by agents belonging to that context.

In this respect, situatedness reinforces even more the fact that compositional links are related to interactions. In the case of conversational MAS, interactions *are* conversations, and the associated contexts are the sets of agents involved in these conversations. Section 3 generalizes this idea by defining conversations as a medium to support dynamic compositional relationships and MAS flexible architectures.

3 Abstracting and Reifying the Architecture

In modern MAS or DOS/DCOS, we usually talk about Open Architectures, since these systems are intended to be easily plugged with external components. Intuitively, the term architecture comprises the idea of stability and durability. It may not be the best word to describe such systems compositional structures *at a given time*, since these structures are supposed to evolve and change — if not always dynamically, at least quite frequently. Here we distinguish the *configuration* which represents the *system compositional structure at a given time*, but subject to change dynamically, from the actual *dynamic architecture*, which is found at a more generic level, and includes mechanisms allowing dynamic composition and dynamic changes.

In such a system, a greater part of the compositional behavior is *reified* and *operationalized* in the system itself, not only through static bindings. In a MAS meeting these requirements, the dynamic architecture will then be partly “agentified”, i.e. determined by the behavior of some agents¹. In next subsection we examine a common paradigm used in this purpose: the directory service paradigm.

3.1 Directory Service Paradigm

The paradigm described here is a very general pattern, and can be found in various systems, usually implemented as a specific solution, or through existing standards such as the Object Trading Service defined in CORBA [2], the Facilitator Agent infrastructure of FIPA architecture [5], or any other directory based solution.

¹ we could substitute the word “agent” with “component” in the case of DCOS.

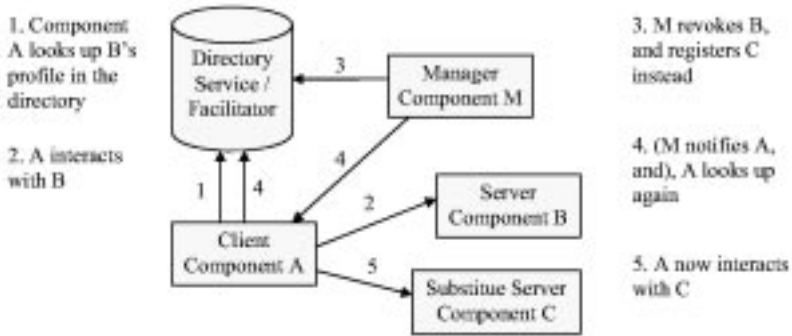


Fig. 2. Component Directory Service or Facilitator Agent Paradigm

The principle is straightforward. In order to manage compositional relationships dynamically, a level of indirection is added: client components or agents must query a *directory service* to get the references of the components they will interact with.

Figure 2 illustrates this *late binding* mechanism. When the system initializes, or when a specific condition is met, client components look up the directory for the server components they are interested in. Some manager components may invalidate the link, and trigger a new lookup by the client components. In MAS, this function is ensured by facilitator agents. Such intermediate can serve several purposes:

- application specific matchmaking, when the targeted service components or agents obviously depend on dynamically available data;
- fault tolerance, ensuring that backup services can be made available dynamically;
- load balancing, finding a sufficiently available component or server;
- and components or agents dynamic substitution.

Some of these functionalities may be implemented at a lower level: for example, load balancing can be managed directly by an Object Request Broker (ORB), however the paradigm remains similar: the ORB routes messages and connects client components to remote components dynamically, using some kind of internal directory. In this respect, we may even think of such architectures, CORBA or EJB, as a kind of directory service based architecture with a very precise service description: the object interfaces, possibly combined with some primary key or object IDs.

At a macro scale, Web Services, served by a UDDI directory and described by WSDL entries, also largely comply with this general pattern, as described in [12].

All these architectures are characterized by the fact that intermediaries are, on the contrary to the components they serve and link, *explicit* and *fixed*: they cannot be introduced or removed dynamically.

Example: Electronic Banking Scenario. To give a concrete example, let's imagine a bank wishing to provide bank account online access to its customers, either through Web Service access, or through dedicated client components. When a client first connects, a main session component is created at the server side. Each time the client

issues a request, the session component searches a specialized directory component for an adequate service component corresponding to the request type.

Since transactions are short, and services stateless, we can imagine that in this case the directory is systematically searched by session components. If a service appears to be out of order, then some backup component may be returned instead by the directory, or the session component may explicitly search for it. If maintenance occurs on a service, a component substitution can be done dynamically without difficulty.

Benefits. Using extensively this paradigm actually reifies the configuration into the Directory Component and what we named Manager Component on figure 2. What is usually put forward by designers of such architectures is that the resulting systems are linked dynamically, very scalable, and the maintenance is significantly eased. There is usually no need to shutdown the system, and component substitution is done at runtime. However this paradigm presents some weaknesses.

Weaknesses. The main problems encountered with such infrastructures are:

- The ignorance of inter-component synchronization, as discussed in 3.3;
- The composite atomicity (transparency), discussed in 2.3, is not verified;
- The directory look up constraint put on client components or agents.

Indeed, defining when client components should look up in the directory may be application specific. In the case of fault tolerance issues, for example, a manager component may pole the services for soundness regularly, and invalidate them if necessary. More complex notification mechanism may also be used.

More generally, the client components or agents have to look up explicitly in a well known directory, which is a constraint because the directory must be fixed.

Finally, we can point out that this paradigm tends to lead to a relatively centralized solution, restricted by the directory capabilities in terms of service description and matching behavior. However, we could also imagine a similar paradigm with multiple, distributed, possibly federated, specialized directories or facilitator agents.

3.2 Conversation Centered Design

In conversational MAS, conversations are at the center of the design, since this notion both represents the interactions taking place among agents, and the temporary roles assigned to agents. A conversation is ruled by an interaction protocol, and an interaction protocol comprises several roles (usually two, possibly multi-bilateral, but multi-party protocols are more and more common and deserve consideration). The roles dictate the expected behavior and responsibilities of each agent taking part of the conversation. A conversation has the following inherent properties:

- it represents a context for a collective task, a situation;
- it has a defined life cycle, and is by nature temporary;
- it is a reusable design pattern by itself, with well defined roles and responsibilities.

If we draw a parallel between roles and interfaces, and between agents and objects, designers could specify the configuration and dynamic architecture of the whole sys-

tem in terms of roles, protocols, agents, in the same way as we do with classes, interfaces and objects. This approach is quite natural with the Unified Modeling Language (UML). UML collaboration diagrams and sequence diagrams are a way to specify protocols. Agent UML [1], for example, uses extended Sequence diagrams to represent FIPA interaction protocols.

An interesting point about conversations is that they are a natural medium for transactions and sessions related concepts, at the agent level. For example, credentials and access rights can be associated with a conversational role, rather than a thread of execution which would not really make sense in such a distributed environment. Similarly, we propose to associate a compositional relationship with a conversation.

3.3 Proposed Alternative Paradigm: Conversational Role Delegation

Definition. We define conversational role delegation as the act of dynamically delegating one agent role, in the context of a target conversation, to another agent, for the duration of the target conversation only. The delegation itself is governed by a simple interaction protocol similar to the FIPA brokering protocol. It has no global or permanent meaning whatsoever, outside the context of the target conversation..

This action is different from the simple act of requesting another agent to do something, which is a weak definition of delegation used sometimes in the literature. Any directive performative implicitly conveys this semantic. For a delegation to occur, an agent has to initially entail the responsibility or role, and corresponding credentials, *with respect to other agent*. An agent cannot delegate something that it is not supposed to do first. Thus we distinguish *delegation*, linked to other agents' perception of the delegator responsibilities, from a mere *request*.

Implementation details. The delegator encapsulates the target conversation initial message(s) to be sent (or received if the delegator is not initiator of the conversation) into a delegation request, and sends the resulting message to the desired delegate.

If the delegate refuses the delegation requests, it resends back the first message(s), encapsulated in a refuse message, to the delegator. Figure 3 illustrates the corresponding interaction pattern. As we will see in section 4.1, an additional message parameter is maintained in the target conversation messages, represented on figure 3 by the sequence (a: (X, D), b: (Y) ...). This parameter represents the delegation chains for each role, and must be updated consistently by agents performing the delegation.

To ensure a correct message delivery synchronization during delegation, and to simplify delegating agents implementation, it seems to us natural, while not obligatory, to implement the message redirection mechanism at the message transport system level of the underlying platform (see section 4.1).

This interaction pattern resembles the FIPA-Brokering protocol, although FIPA-Brokering is more specific and restrictive. In section 4.1 we detail the integration of this mechanism into the FIPA environment.

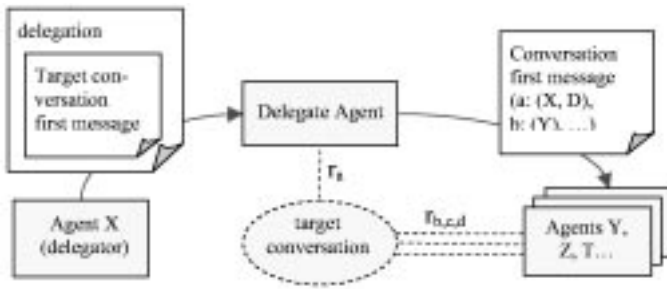


Fig. 3. Conversational Role Delegation Principle

Seamless, Transparent and Situated Dynamic Composition. Conversational role delegation allows building dynamic architectures verifying the requirements identified in section 2.3. This requires the use of specific agents, which we designate by *Composition Managers*, personifying the delegation strategies and compositional behavior of the sub-system they represent. Interaction protocols also need to be properly defined, so that independent roles and responsibilities are adequately distributed among well defined roles or sub-roles in the organization.

Figure 4 illustrates an arbitrary initial configuration of such a MAS dynamic architecture. The composition managers assign roles by delegating them to the appropriate agents. Some may cascade role delegation to other agents seamlessly, as does M2 with agent 4 in the figure. Meanwhile, delegates may be involved in other conversations: for example, in figure 4, M2 manages both r_2 and r'_2 in C and C' respectively, thus M2 manages the mapping between two sub-systems.

Manager agents serve as transparent intermediate agents, and represent the indirection level. They are responsible for the dispatching of the roles they are initially responsible for, by delegation, and may possibly require the help of external agents, facilitator, or use any private strategy to determine the correct role assignment. Such strategies may support a higher level organization. The important point is transparency: *no constraint is put on the client agents.*

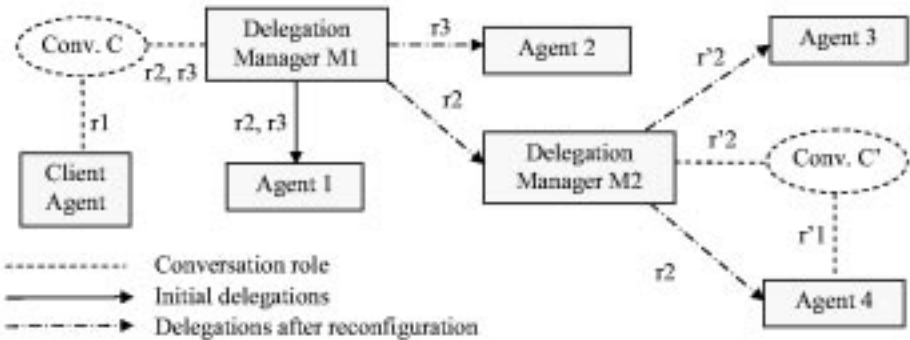


Fig. 4. Example of an initial configuration with a delegational architecture

Seamless dynamic composition is achieved by delegating roles or sub-roles to specific delegates, for each conversation occurrence. Delegation Managers agents have a function equivalent to specialized facilitator agents in the directory service paradigm; however they are seen by other agents as initially responsible for the concerned roles or services. Recursive decomposition is easily done by cascading delegations.

Similarly, seamless decomposition is achieved by substituting a single agent with “composite agent” sub-system — which in turn will be represented by a delegation manager, yet this is transparent to the initial delegation manager.

The compositional links and structure of the system is determined by the delegation strategies and behavior of the different delegation managers. Agent situatedness is also verified: when an agent delegates a role, only the concerned agents, in the context of the concerned conversation, are affected. Thus, the delegate may be involved in another delegation, without interferences with the former one.

This agent oriented design pattern has similarities with the *Abstract Factory* design pattern, as well as the *Composite* design patterns, described in [4], with the important additions that it validates the *situatedness* and *transparency* properties.

Synchronization with ongoing interactions. Role delegation provides a natural support for synchronization because a delegation is by construction attached to a conversation life cycle. The redirection mechanism ensures that no messages will be lost or read twice by agents and their delegates. Conversations semantically represent consistent collective tasks, which may have a transaction semantic too, and they are guaranteed not to be broken by unsynchronized agent substitution.

Regarding this issue, the directory or facilitator paradigm has two weaknesses:

- first, the role assignment is managed by another agent, not necessarily synchronized with the participating agents: asynchrony problems may arise if for example an assignment is changed during the initialization of a complex conversation, where multiple participants are looking up for the reassigned role concurrently;
- second, in the case of cascaded facilitators, which would probably be inevitable with large systems, the propagation of a new role assignment may lead to inconsistent state during the change, unless a synchronization mechanism is used;

Such problems are avoided with delegation, because it is associated with a conversation, and because the agent responsible for the role assignment is initially involved in the conversation, and thus has the necessary knowledge and control to keep it consistent and synchronized.

Protocol composition. Protocol composition is an important in our approach, since it inform us about the structure of interactions, which greatly helps dynamic composition: it shows when conversations or roles can be cut, through the definition of sub-conversations, in order to substitute or aggregate agents. This structuring implicitly defines some sort of conversational quiescent states for the participants.

Unfortunately, there is currently no satisfying theoretical background, to our knowledge, formalizing clearly protocol and conversation composition. Some theoretical properties that would be of interest are:

- Role multiplicity; the conditions and constraints applying to roles assignable to multiple agents simultaneously, are not clearly stated in the current formalisms;
- Protocol composition data, such as the role mapping between a protocol and a sub-protocol, and the possible state synchronization between corresponding roles, is poorly represented in current formalisms: a more formal definition would help.

Other Benefits. In terms of messages exchanged, delegation is more optimized than directory look up. This may be appreciated if there is a high communication delay.

Finally, delegation allows encapsulating the compositional behavior and data of a subsystem into that subsystem, which result in a distribution following the system organization. In the case of the directory paradigm, even if directories are federated, their distribution will not necessarily map the system inherent structure.

4 Scenarios and Prototype Implementation

4.1 Integration into FIPA Environment

The reader can refer to [6] for a detailed description of the concrete integration into the FIPA architecture and messaging system (the current section is a brief summary). This integration relies on two extensions of the FIPA messaging model:

1. The `conversation` parameter, an additional message parameter meant to extend or replace the existing `reply-to` parameter; this parameter is a list of roles associated to a delegation chain (a list of agent IDs), the first in the list being the originator of the role. It should be updated by agents whenever they perform a delegation, or receive an updated version of the parameter.
2. An automatic redirection mechanism at the Message Transport System level, providing agents with a platform native method call to set up a redirection for the duration of a conversation, and associated to a given role.

This mechanism is not obligatory, since agents can redirect themselves subsequent message received after the delegation. However factorizing and moving this function into the platform simplifies agent implementation, because it is meant to be used frequently in an architecture where each conversation may imply delegations. Additionally, if agents are very distant, the different platforms may provide some redirection optimizations (by redirecting messages directly to the last delegate, for example).

4.2 Electronic Auction Scenario and Prototype

Electronic auctions may not be the most illustrative scenario of the actual benefits of such architectures, since they already include dynamic agent aggregations and substitution in the auction process: new auctioneers may join or quit, some may hire brokers, etc. This kind of actions seems natural in the auction context.

Having said this, auctions are also very common in MAS test-beds, which makes them a good starting point, at least regarding the feasibility of the approach.

Scenario and Prototype Description. In this scenario we consider the problem of protocol adaptation to demonstrate the feasibility of conversational role delegation in the FIPA environment. A seller agent implements a simple point-to-point negotiation protocol. The latter delegates the auction initiator role to one adaptation proxy, on the fly, depending on the intended target auction protocol. Additionally, participants implementing incompatible auction protocols may take part of the auction, thanks to “participant-side” adaptation proxies. Several configurations with the following FIPA auction interaction protocols have been tested:

- FIPA-Contract-Net protocol, a single-round one-to-many negotiation;
 - FIPA-Dutch-Auction protocol, an iterative auction with a decreasing price;
 - FIPA-English-Auction protocol, similar but with an increasing price;
 - FIPA-Iterated-Contract-Net protocol, an iterated version of Contract-net.
- An additional one-to-one negotiation protocol, with two-round confirmation, is used as a generic protocol for the seller agent, to adapt easily to auction protocols.

Proxies have two functioning modes: either they create internal Task components for each adaptation conversation, as the standard FIPA-OS componential approach would suggest; or they act as “factory” delegation managers, instantiating new dedicated proxies on demand. Two participant-side proxies have also been designed to adapt Dutch and English auctions participants to Contract-Net conversations.

Initiators and participants agents use a default delegation behavior component, and search for adequate delegates when faced with protocol incompatibility. Participants also advertise this property to the DF, since our DF is able to check for indirect mappings, based on the availability of adaptation proxies.

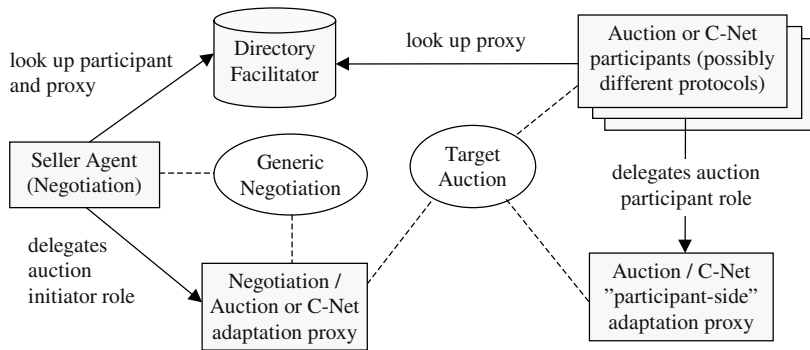


Fig. 5. Prototype Initial Configuration (Flexible Architecture). Dotted lines show *potential* conversation involvements, and plain arrows show *potential* look up and delegations

Architecture and Implementation. This prototype is build as an extension of the FIPA-OS platform [5]. This extension consists of a message redirection mechanism inserted in the transport message system, and the extension of the message class to include the `conversation` parameter. The Directory Facilitator implementation has been extended to account for indirect mappings, through any declared adequate adaptation proxy.

Figure 5 illustrates the initial configuration, with all potential conversation involvements and delegation among agents. In the figure, only one proxy (of four), one “participant-side” proxy (of two), and one participant agent (of four) are represented.

To adapt protocol roles, not only does the delegator delegates its role, but it also starts a conversation with the latter using the protocol implemented initially; both conversations occur at the same time.

Results and Comments. The first result from this prototyping is that the interactions and mechanisms described in this approach are feasible, and can be integrated into the FIPA environment, or similar ones like KQML-based platforms.

The second result is a direct comparison between the classical componential approach used in FIPA compliant agent development platforms (the first proxy functioning mode), and our approach (the “factory” mode), in terms of re-usability, composability, and code complexity. Dedicated proxies are equivalently complex to implement compared to their corresponding FIPA-OS Task component, while bringing a significant benefit in terms of reusability, interoperability and composability. Moreover, the implementation is closer to the multi-agents philosophy, since complexity is expressed by agent interactions rather than intra-agent complex component composition: this approach encourages more agents, but simpler to implement.

5 Conclusion

In this paper we have presented a paradigm to build dynamic architectures in the context of conversational multi-agents systems. This work also concerns distributed object based and component oriented systems, although in these systems the notion of conversation is not as present as with MAS. A comparison with a commonly used paradigm, which we have been referring to as the component directory or facilitator agent paradigm, allowed us to exhibit significant advantages of our approach, in the case of reconfiguration with complex interaction protocols and interdependent services integrated into synchronized collective tasks (the conversations). This paradigm should be seen as a complement, and not a replacement, to the facilitator paradigm — there are of course situations where the latter is more adequate.

The main point is that the notion of software architecture itself is not to be taken as static compositional relationships in modern systems, including MAS, but as specific environments hosting dynamic configurations. In order to design such systems effectively and allow a good scalability, designers should be able to tune the levels of indications, the degree of dynamicity, as well as other non-functional aspects, without affecting the functional design, and with a minimal impact on running agents.

Multi-agents oriented software engineering must rely on constructs that respect the fundamental properties of MAS, including dynamicity, agent situatedness and autonomy. We argue in this paper that it should also somehow bring composability at the agent level. Conversation centered design, and conversational role delegation, allow designing architectures that fulfill these requirements. Therefore they should be considered as important aspects of agent platforms and design methodologies.

The main perspective of this research is to extend our current prototype to a more illustrative and complex scenario, where the system is subject to dynamic architectural upheavals. Other perspectives are to rely on a more formal theory and model of interaction protocols composability, accounting for multi-party protocols, among other things. Although efforts exist in this direction, such a theory is to our knowledge still missing or immature.

While primarily centered on the idea of a meaningful compositional relationship at the agent level, our work definitely implies a rather holonic view of MAS, and a role oriented modeling approach where roles are dynamically assignable, as discussed respectively in [12] and [10] in these proceedings.

References

- [1] Bauer, B., Müller, J., Odell, J.: Agent UML: A Formalism for Specifying Multiagent Interaction. In proceedings of Agent-Oriented Software Engineering 2000, Ciancarini, P., and Wooldridge, M., eds., Springer LNCS, Berlin (2001)
- [2] OMG: Common Object Request Broker Architecture (CORBA) specifications v. 3.0 (Component Model, Object Trading Service). <http://www.omg.org/technology/documents/>
- [3] Magee, J., Kramer, J., Sloman, M.: Constructing Distributed Systems in Conic. In Transactions on Software Engineering, 15 (6), IEEE (1989)
- [4] Gamma, E. , Helm, R. , Johnson, R., Vlissides, J.: Design Patterns, Elements of Reusable Object-Oriented Software. Addison Wesley (1995)
- [5] Foundation for Intelligent Physical Agents (FIPA): FIPA specifications (Abstract Architecture, Agent Management, Message Transport System, and the different Interaction Protocol specifications). <http://www.fipa.org/specification/>
- [6] Jouvin, D., Hassas, S.: Flexible Multi-Agent System Architecture using Dynamic Delegation. In proceedings of the 6th International Symposium on Programming and Systems (ISPS), Algiers (2003)
- [7] Bellifemine, F., Poggi, A., Rimassa, G.: JADE: a FIPA 2000 Compliant Agent Development Environment. Proceedings of the ACM International Conference on Autonomous Agents, Montréal, Quebec, Canada (2001)
- [8] Parunak, H. V. D., Odell, J.: Representing Social Structures in UML for Agent-Oriented Software Engineering. In proceedings of Agent-Oriented Software Engineering 2001, Wooldridge, M., Weiß, G., Ciancarini, P., eds., Springer LNCS, Montreal, Canada (2001)
- [9] Jouvin, D., Hassas, S.: Role Delegation as Multi-Agent Oriented Dynamic Composition. Proceedings of Net Object Days (NOD), AgeS workshop, Erfurt, Germany (2002)
- [10] Odell, J., Parunak, H.V.D., Brueckner, S., Sauter, J.: Temporal Aspects of Dyanmic Role Assignment. (In proceedings of AOSE 2003)
- [11] Poslad, S., Buckle, P., Hadingham, R.: The FIPA-OS Agent Platform: Open Source for Open Standards. Proceedings of the 5th International Conference and Exhibition on Practical Application of Intelligent Agents And Multi-Agent Technology, Manchester, UK (2000)
- [12] Giret, A., Botti V.: Towards a recursive agent model for an agent orient methodology. (In proceedings of AOSE 2003)

- [13] Stal, M.: Web services: beyond component-based computing. Communication of the ACM, Volume 45, issue 10, ACM Press (2002) 71–76
- [14] Nwana, H., Ndumu, D., Lee, L., Collis, J.: ZEUS: a Toolkit and Approach for Building Distributed Multi-Agent Systems. Proceedings of ACM International Conference on Autonomous Agents, Seattle, USA (1999)

Temporal Aspects of Dynamic Role Assignment

James J. Odell¹, H. Van Dyke Parunak², Sven Brueckner², and John Sauter²

¹ James Odell Associates, 3646 West Huron River Drive, Ann Arbor
MI 48103-9489 USA
email@jamesodell.com
http://www.jamesodell.com

² Altarum Institute, 3520 Green Court, Suite 300, Ann Arbor,
MI 48105-1579 USA
{van.parunak, sven.brueckner, john.sauter}@altarum.org
http://www.erim.org/{~vparunak, sbrueckner}

Abstract. A helpful abstraction of a group of agents is a set of interacting roles, or sets of normative behaviors, that the agents can assume. An important characteristic of real-world agent systems is that the roles played by an agent may change over time. These changes can be of several different kinds. We describe an illustrative application where such role changes are important, analyze and classify the various kinds of role changes over time that may occur, and show how this analysis is useful in developing a more formal description of the application.

1 Introduction

The notion of *role* is fundamentally a thespian concept. As humans, we find the perspective and language of the theater a useful analogy for describing and understanding many of the same complex aspects of individual behavior [Odell, 2003]. Since we commonly employ the notion of role in real life for conceptualizing human behavior, it may also serve as a useful device for other kinds of individuals in a MAS—be they life forms, active software constructs, or hardware devices.

In an agent-based system, we define *role* as a class that defines a normative behavioral repertoire of an agent. Roles provide both the building blocks for agent social systems and the requirements by which agents interact.¹ Each agent is linked to other agents by the roles it plays by virtue of the system's functional requirements—which are based on the expectations that the system has of the agent. The

¹ Several possible implementation techniques exist for implementing programs that support social entities possessing multiple and changeable class-based roles, including class inheritance and aggregation. In this paper, we will not discuss program-level implementation options for treating role as a class. Instead, our emphasis will be at the analysis-level (i.e., a conceptual and implementation-independent approach).

static semantics of roles, role formation and configuration, and the dynamic interactions among roles has been examined closely in recent years [Ferber et al. 2003][Castelfranchi, 2000][Destani, 2003][Odell, 2001][Parunak, 2001]. However, little work has been done on formalizing the temporal aspects of dynamic role assignment. As a result, role modelers refer only informally to actions such as “taking on a role,” “playing a role,” “changing roles,” and “leaving roles.” However, such terms can be interpreted ambiguously.

For example, consider the scenarios that follow. These six scenarios are considered to be “changes,” yet semantically they all have a different meaning.

1. *Classify* – Add the role of manager to the role of Employee as the result of a promotion.
2. *Declassify* - Remove the role of Manager as the result of a demotion.
3. **Reclassify* - Change from the role of Employee to the role of Unemployed Person.
4. *Activate* - Take up the behaviors of the Manager role as part of the day-to-day business activity.
5. *Suspend* – Stop any Manager behavior and take on just those of the Employee role as part of the day-to-day business activity.
6. **Shift* - Change from an Employee role to a Pet Owner role, where neither is played at the same time as the other. This is a combination of activating one role while suspending another.

(*These are composite roles, not primitives.)

These scenarios will be discussed in more depth in the following sections. Prior to that discussion, however, some fundamental notions need to be identified and defined.

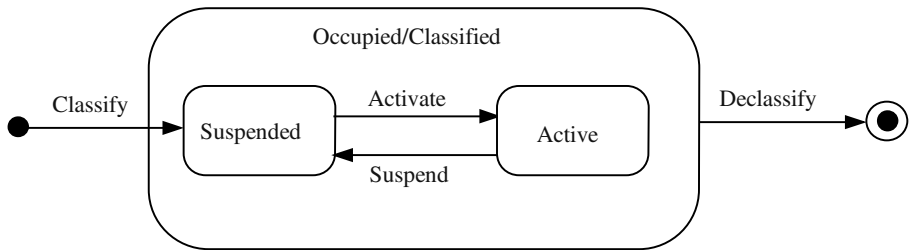


Fig. 1. Statechart depicting some of the permitted states and transitions of an agent in a role.

The statechart in Fig. 1 depicts four of the operations in the scenarios above. Here, an agent comes to *occupy* a role when it is *classified* and ceases to occupy the role when it is *declassified*. Furthermore, while an agent occupies a role, it can be either *active* or *suspended* in playing that role. Transitioning between those two states can be achieved via the *suspend* and *activate* operations. Definitions for these states and transition operations are as follows:



States

Occupied – The state of an entity that is an instance of a particular role.

Suspended - The state of an entity occupying a particular role, when the role has no processes executing.

Active - The state of an entity occupying a particular role that is executing some of all of its processes.

Operations

Classify – An operation that results in an agent being an instance of, or occupying, a particular role.

Declassify – An operation that results in an agent no longer being an instance of, or occupying, a particular role.

Reclassify – An operation that results in an agent being both declassified as one particular role and classified as another.

Activate – An operation that results in an agent entity that is active.

Suspend – An operation that results in an agent entity that is suspended.

Shift – An operation that results in an agent entity becoming both suspended within one occupied role and active in a different occupied role.

These “role-changing” operations can best be discussed by partitioning into categories of dynamics: *dynamic classification* and *dynamic activation*. The sections that follow will describe these notions in more detail. AUML notation will also be proposed.

For concreteness, we develop these ideas in the context of a specific multi-agent application, a swarm of unpiloted air vehicles (UAV’s) that must coordinate their actions in various ways to accomplish a collective sensing task. Section 2 describes this application informally. Sections 3 and 4 develop and refine two aspects of dynamic roles: dynamic classification and dynamic activation, and suggest AUML representations for them. Section 5 applies these refinements to the UAV application, and Section 6 summarizes our contribution.

2 An Example Application

We demonstrate the temporal role dynamics in MAS in an example of a swarm of small robotic air vehicles (UAV) that perform a sensing mission in an urban environment. The mission proceeds through the *deployment phase*, in which the individual units are initialized and released, the *target approach phase*, where the swarm moves through the urban environment and establishes a line-of-sight communications network that links the base with the target area, the *objective completion phase*, where the swarm performs its sensing task and communicates the resulting data through the dynamic network to the base, and the *recovery phase*, where the swarm returns to the base or dissolves depending on the type of UAV (disposable or not). We chose the target approach phase for our detailed discussion of dynamic role assignment.

The mission as outlined in the previous paragraph cannot be performed by a single UAV (agent), because of the restricted communications and sensing capabilities of the units. The restriction to line-of-sight communications in the urban environment requires that the swarm dynamically forms a communications network made up of strategically placed perching or hovering units that route data to and from the deployment base. These stations in the network are assigned dynamically as the swarm proceeds towards its target while avoiding obstacles and threats. Thus, one role an agent may assume during the mission is that of a *Communications Node*. All UAVs in the swarm are physically capable of performing this function.

Those units that do not take on the role of a Communications Node eventually arrive at the target area, ready to perform the mission objective. At this point, the swarm enters the next phase of the mission as some units take on the *Sensing Node* role and coordinate with other Sensing Nodes to perform the sensing task specified in the mission. Only a subset of all the UAVs in the swarm have the more expensive sensor packages that are capable of performing this task. Data acquired by the Sensing Nodes is sent to the base through the network formed by the Communications Nodes.

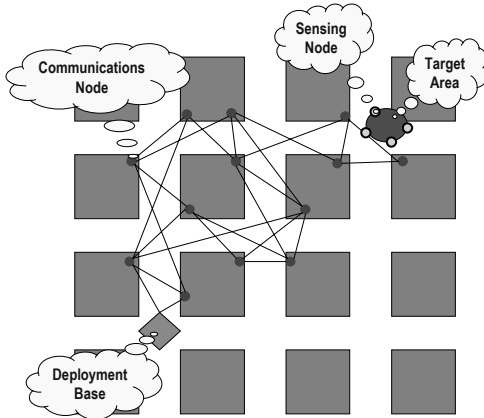


Fig. 2. A Mission Swarm Dynamically Assigning Communications and Sensing Node Roles.

The mission swarm operates in a hostile environment and individual units are prone to failure. Thus the communications network as well as the objective execution team has to cope with continuous attrition. An individual unit is aware of the current state of the communications/sensing team in its local environment and if it determines that attrition in its neighborhood threatens the performance of the mission, either by disrupting the information flow between the base and the target area or by removing required sensing capabilities, it seeks to recruit “uncommitted” units to fill the gap. This recruitment attraction draws units of the initially released swarm as well as additional units from the deployment base. The self-healing mechanism requires that agents are not only aware of the presence of other units in their neighborhood, but also that they know, which role they currently perform and which roles they are physically capable of performing.

3 Dynamic Classification

Dynamic classification refers to the ability to change the classification of an entity. While dynamic classification applies to both object classes and agent classes [Odell, 1998], in this paper we will discuss it in terms of agent roles. For example in Fig.

3(a), the “Alice” agent changes from being an instance of the roles Employee, Manager, and Salesperson to being an instance of the role Unemployed Person. In other words, with dynamic classification, an agent can be an instance of different roles from moment to moment.

Consistent with [Ferber et al. 2003], we insist that each agent have at least one role at all times. Dynamic classification deals with adding additional roles or removing roles beyond the minimum of one. This requirement is analogous with the notion that every human must play the “person” role, whatever other roles they may have. In the case of humans, this minimal role persists throughout the agent’s life. It is conceivable that an artificial agent might begin with the minimal role A, add role B, then remove role A, leaving it with the minimal role B. Whether such a fundamental re-definition of the agent is possible will depend on such features as physical equipment associated with the agent and the nature of the platforms on which the agent can run.



Fig. 3. (a) Dynamic classification refers to the ability to change the classification of an agent’s role. (b) The agent “Alice” moving in and out of being classified as an Employee over time.

To become an instance of a given role, the agent is *classified* as an instance of, or *occupies*, that role. Once classified, then, the agent occupies the new role and possesses all of its features. In the opposite process, if an agent is *declassified*, it is removed as an instance of a particular role—and no longer occupies the role nor possesses features unique to that role. Figure 3(b) portrays the “Alice” agent being classified and declassified in terms of the role Employee. At some point in her life, Alice is first classified as an Employee. Later, through some process, Alice is declassified as an Employee: she becomes unemployed. At another point, Alice may become reemployed, followed again by a period of unemployment. This behavior may continue until retirement is reached or the process of death takes place. And where both operations are used at the same time, an agent is said to be *reclassified* when it is both declassified in one role and classified as another.

Based on the descriptions above, we can now discuss “change” scenarios 1, 2, and 3 without ambiguity:

1. **Classify - Add the role of manager to the role of Employee as the result of a promotion.** In this situation, the person still remains an instance of Employee role. However, the person becomes a new instance of the Manager role. In other words, the person remains classified as an Employee but is now—in addition—classified as a Manager. This is called *classification*.

2. **Declassify - Remove the role of Manager as the result of a demotion.** Here, the person still remains an instance of Employee role, but is no longer an instance of the Manager role. *Declassification* has the opposite effect of classification in scenario 1, above.
3. **Reclassify - Change from the role of Employee to the role of Unemployed Person.** In this scenario, the person ceases to be an instance of the Employee role and becomes an instance of the Unemployed Person role. In other words, the person was *declassified* as an Employee, while simultaneously being *classified* as an Unemployed Person. This is called *reclassification*.

The table below summarizes the role assignments involving dynamic classification: classification, declassification, and reclassification operations.²

Operation	Pre-state	Post-state
Classify	A and not B	A and B
Declassify	A and B	A and not B
Reclassify	A	B

Both UML and AUML notations would express these three operators as illustrated in Fig. 4. Figure 4(a) indicates agent-1’s classification as role-*n*; in Fig. 4(b), agent-1 is declassified as role-*m*; in Fig. 4(c), agent-1 is being reclassified from role-*m* to role-*n*.

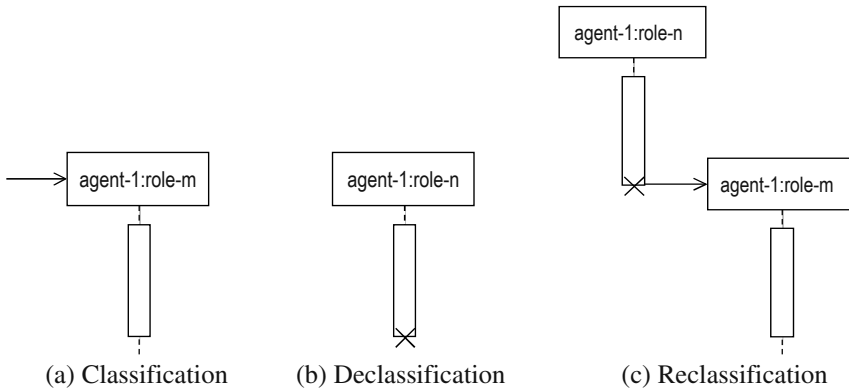


Fig. 4. AUML notation for classification, declassification, and reclassification.

4 Dynamic Activation

In the previous section, we have seen how “Alice” may be classified and declassified as Employee and Unemployed Person roles over time. Orthogonally, she may get married and become an instance of the role Married Person and Wife. Additionally, she

² For completeness, arguably two more operators could be added: *create* and *terminate*. The create operator classifies entities that did not previously exist. Here, the prestate for an entity would be null, and the poststate a particular role. Termination is the opposite, where the prestate would be for an entity in some role and the poststate would be null.

may be confirmed as a Supreme Court Justice or buy a pet and become a Pet Owner. While she may be a Supreme Court Justice for life, she may later give the pet away and be removed from the Pet Owner set. Every agent, then, must always be classified in at least one role — where Agent can be thought of as a role in its own right.³

In her lifetime, Alice may be an instance of many roles. This means, first, that the roles that apply to an entity can change over time (dynamic classification). Second, it means that an entity can have multiple roles that apply to it at any one moment. When an entity is an instance of more than one role this is called *multiple classification* (not to be confused with multiple inheritance).

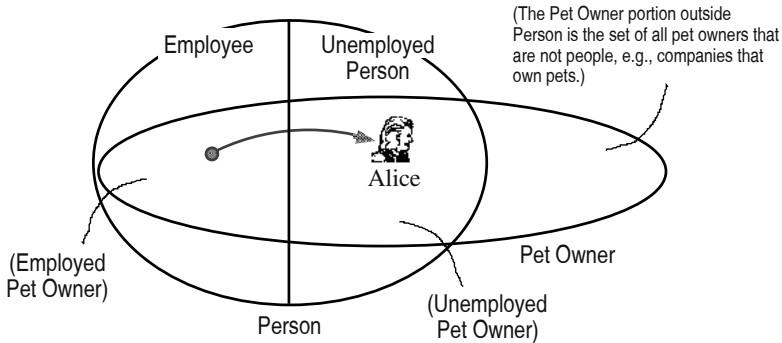


Fig. 5. “Alice” involved in both dynamic classification and multiple classification.

Multiple classification is a useful notion here, because in a society-based system an agent is likely to be active (or quiescent) in multiples roles at the same time. However, what does it mean for a role to be “active”? In some sense, a quiescent agent that is waiting for a message or some signal to awaken could be considered active in its role, because alertness can be thought of an activity. UML 2.0 [OMG, 2003] offers a useful refinement by distinguishing between user-defined actions (which are represented explicitly in sequence diagrams and activity diagrams) and fundamental system actions such as *i/o*, invocation, and data flow (which are not represented as actions in these diagrams). In UML, each activation, or *execution occurrence*, has some duration and is bounded by a start and stop point. We propose to take advantage of this refinement in the following unification:

- We adopt the UML 2.0 definition of action. Any unit of behavior that has started and has not yet ended is considered “active.” Otherwise it is “inactive.”
- We use the basic role of *AgentId* to specify primitive behavior. (Id in *AgentId* is meant to suggest the Freudian sense of primal basic urges, not the sense of “Identity.”) Behavior such as controlling, handling data flows, and waiting for messages and signals can be thought of as “primitive” actions that all entities must possess to be agents. Therefore, any entity playing the role of *AgentId* can exhibit this basic

³ Having Agent as a role is a controversial point. However, in [Odell, 2003] we defined *role* as a class that defines a normative behavioral repertoire of an agent. The basic class called Agent defines the normative behavioral repertoire for agenthood.

behavior, deferring “higher-level” behavior to user-specified actions in more specialized roles. Furthermore, these basic behaviors are themselves actions. For example, actions that support listening for messages and signals, *by definition*, begin the moment an entity is classified an *AgentId* and cease when the entity is no longer an *AgentId*. This default role satisfies the criterion by Ferber and Gutknecht [Ferber, 2003] stating that “every agent plays at least one role,” in this case the *AgentId* role.

- *We consider roles other than AgentId to be active only when their user-defined actions are active.* Activity of primitive actions is attributed to the concurrently executing *AgentId* role, not to the user-specified role.

Dynamic activation involves the following operations: activate, suspend, and shift. These operations are discussed in “change” scenarios 4 through 6, as follows:

4. **Activate - Assume the behaviors of the Manager role as part of the day-to-day business activity.** Here, the person still remains an instance of both Employee and Manager roles. However, when dealing with the employee’s boss, the person is *active* in playing Employee role. Yet, when the employee starts addressing her subordinates, she also becomes active in the Manager role. In other words, the person begins the scenario as active only in the Employee role, and ends being active in both the Employee and Manager roles.
5. **Suspend - Change from the role of Manager to the role of Employee as part of the day-to-day business activity.** As with the previous example, the person still remains an instance of both Employee and Manager roles. In this scenario, the person might start the day as a Manager role by approving an expense report for a subordinate. However, if the person then reports to her boss, she *suspends* her role as Manager. In other words, the person begins the scenario as classified and active in both Employee and Manager roles, and ends with the Manager role suspended while the Employee role remains active. (This is the opposite effect of the scenario 3, above.)
6. **Shift - Change from an Employee role to a Pet Owner role, where neither is played at the same time as the other.** Here, the person still remains an instance of both Employee and Pet Owner roles. However, this differs from scenarios 4 and 5, because the person is not active in both roles at the same time. Here, the person suspends his Employee role and becomes active in the Pet Owner role—yet consistently remains classified in both Employee and Pet Owner roles.

The table below summarizes the role assignments involving activation-related classification: activate, suspend, and shift.⁴

⁴ Four combinations were omitted from this table. The situations where the pre- and post-states have only suspended roles and where the pre- and poststates have only active roles is not interesting here because there are no state changes. The prestate, where only active role(s) exist and become only suspended ones, is just two concurrent cases of suspension. The prestate, where only suspended role(s) exist and become only active ones, is just two concurrent cases of activation.

Operation	Prestate		Poststate	
	Active	Suspended	Active	Suspended
Activate	A	B	A and B	
Suspend	A and B		A	B
Shift	A	B	B	A

Both UML and AUML notations would express these three operators are illustrated in Fig. 6. Figure 6(a) indicates agent-1 is activated as role-m; in Fig. 6(b), agent-1 is suspended as role-m. In Figs. 6(c) and 6(d), the role of agent-1 shifts from role-n to role-m. Asynchronous messages (indicated by the open arrowhead) do not require a response. Therefore, the shift would proceed from the end of an execution occurrence bar (the thin triangle over the lifeline) for one role to the beginning of the execution occurrence bar for another (Fig. 6(c)). In contrast, synchronous messages (indicated by the solid arrowhead) require a response before proceeding. Figure 6(d) depicts an agent in role-n sending a message that activates role-m. Since the message is synchronous, all role-n processing is suspended until control is returned from role-m (the dashed arrow).

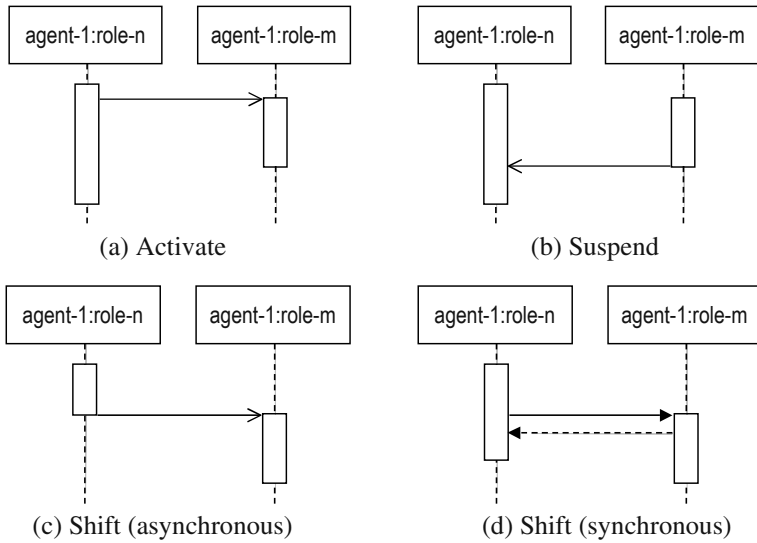


Fig. 6. UML and AUML notation for expressing activate, suspend, and shift.

For graphical clarity, message lines can be supplemented with stereotypes. For the activate, suspend, and shift operations, the stereotypes would be «activates», «suspends», and «shifts».



5 The Application Analyzed

Let us now revisit the sensing mission example (Section 2) in the light of the formal specification of role dynamics. At the moment of deployment, the units with the sensor package occupy three roles (Task Controller, Flight Controller, and Sensor) and the units without the sensor package occupy three roles (Task Controller, Flight Controller, and Communications Node).

The *Task Controller* (TC) role is activated when the UAV is first turned on. It is responsible for dynamically determining the current tasks that the unit performs during the mission. It uses a set of role activation and transition rules to determine what task it should be performing.

The role of *Flight Controller* (FC) specifies behaviors needed to control the movement of the UAV such as path planning, collision avoidance, etc. The Task Controller will initially inform the Flight Controller of the target location and the Flight Controller will begin to move the UAV towards the target area.

The *Sensor* role manages the task of collecting the target data and transmitting the results back to the deployment base station. When it is activated, it may give new flight coordinates to the Flight Controller so it can perform its function.

The *Communications Node* (Comms) role, causes the UAV to maintain line-of-sight with nearby neighbors and relays communications packets between the base station and the rest of the swarm. It also sends movement commands to the Flight Controller.

The transition rules (in the Task Controller role) are context dependent (e.g., where is the agent, what is its state, what is the locally perceived demand for certain functions within the swarm) and constrained by the unit's physical capabilities (e.g., available sensors, remaining fuel) and individual preferences (e.g., risk aversion, susceptibility to peer-pressure). The rules combine the current context, constraints, and preferences to determine current active role(s) as well as whether to declassify current roles and classify new roles.

For instance the Task Controller role in a UAV may perceive a lack of UAVs with active Communications roles within its line of sight and choose to activate its Communications role⁵. Once the Communications role is activated it maintains line of sight with its nearest neighbors and relays messages. Other UAVs in the vicinity, perceiving the newly activated role, continue on their original flight plan towards the target area realizing that the communications relay function is sufficiently covered in that area now. The Communications role remains active until the mission is complete and the UAVs begin their egress back towards the deployment base. Once its position as a relay point is no longer required to maintain communications with the swarm, it suspends the Communications role and the Task Controller instructs the Flight Controller to head home. Figure 7 shows the AUML diagram for this UAV (not all the interactions are shown on the diagram). A similar diagram could be developed for the

⁵ The Task Controller uses a probabilistic decision process to determine whether to activate the role. The exact nature of the decision process is beyond the scope of this paper.

nodes that activate their Sensor role when they reach a target area to coordinate with other UAVs collecting data on the target.

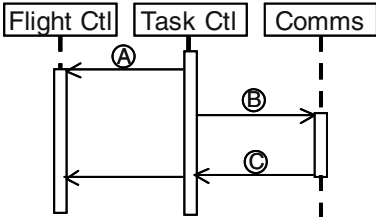


Fig. 7. AUML for a Communications UAV. A: TC instructs FC to move to target area. B: Enroute, TC determines need to stop and serve as communications relay. C: Mission completes, Communications role is done, and TC tells FC to fly home

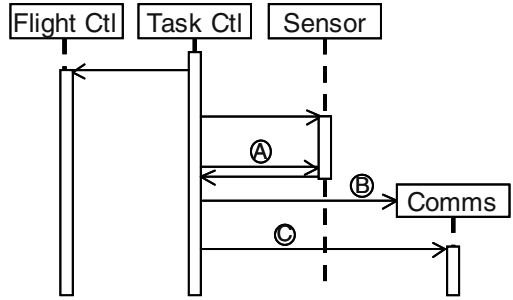


Fig. 8. Classification. A: TC determines a need to change to a Communications role, so it suspends the Sensor role, B: Classifies the Comms role, and C: activates the Comms role.

Since there are only a few UAVs with sensor packages, they would normally not be used to serve as communications relays. But in an emergency they can perform that function. Since they do not initially occupy the role of Communications Node, they must change their role through a classification operation. Figure 8 depicts one of these UAVs that was in the middle of a sensing role and determined that there was a higher priority need to fill a Communications Node role. It suspended the Sensor role, classified the new role of Communications Node, and activates that new role. Classification in this case is more than just activation of previously existing code in the UAV. A communications relay must have current routing information. The exact form of this information varies with the communications protocol in use, but the information is dynamic and in the nature of the case must be learned by a relay UAV before it can perform this function. Classification is only possible for a UAV that has

learned this information (perhaps by lurking on the network for a while without relaying, or by requesting routing information from a nearby routing UAV and using this as a first approximation). Part of what the Task Controller does in the process of classifying the UAV for the relay role is acquiring this routing information.

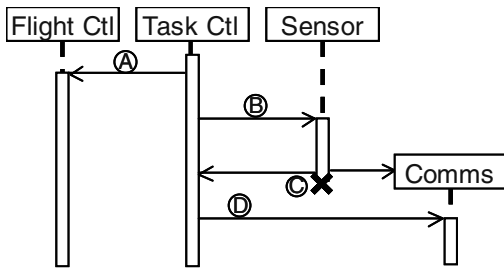


Fig. 9. Reclassification. A: TC instructs FC to fly to target area B: UAV reaches target and TC activates Sensor role C: Sensor HW failure, reclassify as Communications Node, D: TC activates a Comms role.

A last example demonstrates reclassification. In Fig. 9 the UAV performing its sensing task (active Sensor role) experiences a hardware failure in its sensor package so it

can no longer perform the sensor function. It immediately reclassifies itself from the Sensor role to the Communications Node role which is later activated by the Task Controller when it determine a need for that role in the area. Similarly if the UAV loses its ability to move (e.g., through hardware failure, or loss of fuel) the Flight Controller role would be declassified and the Task Controller would probably activate the Communications role (if not already active) to serve the mission in its only remaining capacity. This may free up a nearby UAV serving as a communications node to move on to perform some other function elsewhere in the swarm.

6 Conclusion

Roles are increasingly recognized as a valuable abstraction for modeling groups of agents. In dynamic environments, an agent may change the roles it plays over time. Analysis of these changes show that they fall into two general categories.

1. The more conventional concept is Dynamic Activation. An agent may incorporate multiple roles but not be active in all of them at the same time. Varieties of Dynamic Activation describe the different patterns in how an agent activates or suspends the various roles that it possesses.
2. Dynamic Classification deals with the more fundamental binding between an agent and a role. Straightforward mechanisms for role assignment in contemporary programming languages (e.g., inheritance from a class defining the role's behaviors) are static and persist for the agent's lifetime. However, the concept of Dynamic Classification encourages us to conceive of roles being bound to an agent after the agent is instantiated, and unbound without terminating the agent.

We have demonstrated the usefulness of these concepts in formalizing the behaviors of a swarm of unpowered air vehicles performing a cooperative sensing task.

The notion of roles invites a new approach to agent programming, in which the unit of agent invocation is the role rather than the individual behavior. In such a role-oriented programming environment (ROPE), invocation consists of passing an agent a role and an execution environment (compare the notion of Agent Coordination Context in [Omicini, 2002]), and it is up to the agent to carry out the role in that context. Development of ROPE is a future opportunity for this line of research.

Acknowledgements. This work is supported in part by DARPA, contract F30602-02-C-0196 to Altarum, under DARPA PM Vijay Raghavan. The views and conclusions in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

The authors are grateful to their colleagues Paul Chiusano, Bob Matthews, Mike Samples, and Peter Weinstein for insightful review and comments on earlier drafts of this paper.

References

- Castelfranchi, Cristiano (2000) "Engineering Social Order," *Engineering Societies in the Agent World*, Springer, Berlin, pp. 1–18.
- Dastani, M., V. Dignum, and F. Dignum (2003) "Role Assignment in Open Agent Societies" in *Proceedings of AAMAS'03, Second International Joint Conference on Autonomous Agents and Multi-agent Systems*. 2003. Melbourne, Australia.
- Ferber, J., O. Gutknecht, et al. (2003). "Agent/Group/Roles: Simulating with Organizations." *Fourth International Workshop on Agent-Based Simulation (ABS03)*, Montpellier, France.
- Martin, J. and J.J. Odell, *Object-Oriented Methods: A Foundation*. UML ed. 1998, Englewood Cliffs, NJ: Prentice Hall.
- Odell, J.J., *Advanced Object-Oriented Analysis & Design using UML*. 1998, Cambridge, UK: Cambridge University Press.
- Odell, J., H.V.D. Parunak, and B. Bauer (2001) "Representing Agent Interaction Protocols in UML," in *Agent-Oriented Software Engineering*, P. Ciancarini and M. Wooldridge, eds. 2001, Springer: Berlin. p. 121–140.
- Odell, J., H.V.D. Parunak, and M. Fleischer (2003), "The Role of Roles in Designing Effective Agent Organizations," in *Software Engineering for Large-Scale Multi-Agent Systems*, A.F. Garcia, et al., eds. 2003, Springer-Verlag: Berlin.
- Omicini, A. Towards a notion of agent coordination context. In D. Marinescu and C. Lee, Editors, *Process Coordination and Ubiquitous Computing*, 187–200. CRC Press, 2002.
- Parunak, H. Van Dyke and James Odell (2001) "Representing Social Structure using UML," *Proc. of the Agent-Oriented Software Engineering Workshop, Agents 2001 Conference*, Paolo Ciancarini, Michael Wooldridge, and Gerhard Weiss eds., Agents 2001 conference, Montreal, Canada, Springer.

From Agents to Organizations: An Organizational View of Multi-agent Systems

Jacques Ferber, Olivier Gutknecht, and Fabien Michel

LIRMM – University of Montpellier II, 161 rue Ada,
34592 Cedex 5, Montpellier, France
{ferber,olg,fmichel}@lirmm.fr

Abstract. While multi-agent systems seem to provide a good basis for building complex software systems, this paper points out some of the drawbacks of classical “agent centered” multi-agent systems. To resolve these difficulties we claim that organization centered multi-agent system, or OCMAS for short, may be used. We propose a set of general principles from which true OCMAS may be designed. One of these principles is not to assume anything about the cognitive capabilities of agents. In order to show how OCMAS models may be designed, we propose a very concise and minimal OCMAS model called AGR, for Agent/Group/Role. We propose a set of notations and a methodological framework to help the designer to build MAS using AGR. We then show that it is possible to design multi-agent systems using only OCMAS models.

1 Introduction

Since their development in the 80’s multi-agent systems have been considered as “societies of agents”, i.e. as a set of agents that interact together to coordinate their behavior and often cooperate to achieve some collective goal. It is clear, from this conception, that the body of multi-agent researches should be concerned by both agents and societies. However, an important emphasis has been put on the agent side. Multi-agent systems have particularly been studied at the micro-level, i.e. at the level of the states of an agent and of the relation between these states and its overall behavior. In this view, communications are seen as speech acts whose meaning may be described in terms of the mental states of an agent. The development of communication languages such as KQML and FIPA ACL follows directly from this frame of mind.

We will use the term “agent centered multi-agent system” or ACMAS for short to talk about this type of classical multi-agent systems designed in terms of agents’ mental states. As we will see in the following section, ACMAS suffer from some weaknesses that cannot be solved at the agent level, because they reside deep in the core of ACMAS foundational principles.

Recently a particular interest has been given to the use of organizational concepts within MAS where the concepts of ‘organizations’, ‘groups’, ‘communities’, ‘roles’, ‘functions’, etc. play an important role [4] [9] [13] [14] [16]. We will call ‘organiza

tion centered multi-agent systems' or OCMAS for short, multi-agent systems whose foundation lies in this kind of organizational concepts.

Thinking in terms of organization design differs from the agent-centered approach that has been dominant during many years. An organization oriented MAS is not considered any more in terms of mental states, but only on capabilities and constraints, on organizational concepts such as roles (or function, or position), groups (or communities), tasks (or activities) and interaction protocols (or dialogue structure), thus on what relates the structure of an organization to the externally observable behavior of its agents. However, while OCMAS might solve, as we will see, the main weaknesses of ACMAS, their characteristics and consequences, have somehow been left out and have not been presented clearly. We will see in this paper, that it is possible to design MAS using only organizational concepts. At first, this approach needs a new state of mind to get away from the agent oriented, now classical, conception. However, it does not mean that agent mental states must be thrown away; we only want to stress that it is possible to build organizations as frameworks where agents with different cognitive abilities may interact. Section 2 will show that some of the weaknesses of ACMAS appear as consequences of the mere foundational principles, somehow implicit, of ACMAS. Section 3 will introduce the main concepts of OCMAS and a set of fundamental principles that could be considered as a kind of manifesto for designing MAS from a pure organizational perspective. In order to show that it is possible to design OCMAS in this framework, we will present, in section 4, a generic but simple organizational model for building OCMAS, called AGR for Agent/Group/Role. This presentation will include the basic concepts and the notation one can use to describe organizations. The remaining sections will introduce a simple example and a sketch of a methodology based on these organizational concepts.

2 Drawbacks of ACMAS

It has been shown that the world of software engineering may benefit from the concepts and architectures proposed by the MAS community [9, 16] in order to simplify the design of complex software systems. In order to make MAS systems ready for industrial applications, a non-profit association called FIPA, has proposed a set of norms and standards that designers of multi-agent systems should meet to make their MAS compatible with other systems. An interesting point about these standards, and the platforms that have been built according to them (see Jade [17] and Fipa-OS for instance [18]), is that they are based on some assumption that lies somewhere in the core of most of early work on MAS.

1. An agent may communicate with any other agent.
2. An agent provides a set of services, which are available to every other agent in the system.
3. It is the responsibility of each agent to constrain its accessibility from other agents.

4. It is the responsibility of each agent to define its relation, contracts, etc. with other agents. Thus, an agent “knows” directly (through its acquaintances) the set of agents with which it may interact.
5. Each agent contains with its name its way to be accessed from the outside (the notion of Agent ID well known by all designers of MAS). Therefore, agents are supposed to be autonomous and no constraint is placed on the way they interact.

In this situation, as Jennings and Wooldridge have been pointed out, ACMAS may suffer some drawbacks when engineering large systems [10], which leads to two major drawbacks, according to Jennings [9]: the patterns and the outcomes of the interactions are inherently unpredictable, and predicting the behavior of the overall system based on its constituent components is extremely difficult (sometimes impossible) because of the high likelihood of emergent (and unwanted) behavior.

Surely, freedom has a price: it is not possible to suppose that agents designed by different designers could interact altogether without any problems. Some assumptions have to be made about the primitives of communications (the “performatives” of the language) and about the architecture of agents (for instance, agents may be assumed to behave purposively in a cognitive way, using some kind of BDI architecture). However, agents do not have access to these constraints that are specified as ISO-like standards, and they do not have the possibility to accept, or refuse, to follow them. This imposes a strong homogeneity on agents: agents are supposed to use the same language and to be built using very similar architectures. The other weaknesses of these MAS are:

1. **Security of applications:** The possibility that all agents may communicate without any external control may lead to security problems. When all agent may interact freely altogether, it is the responsibility of agents (and therefore of the application designer) to check the qualification of its interlocutors} and to implement security controls. Because there is no “general” security management, it is easy for an agent to act as a pirate and use the system fraudulently.
2. **Modularity:** in classical software engineering, entities that closely work together are grouped into modules or “packages”. For each module, rules of visibility are defined. Some entities may be seen by other packages (and even by the whole software) whereas others, so called *private* entities, are hidden and therefore not accessible from outside the package. This is not possible with AOMAS where all agents are accessible from everywhere. It should be important to propose a way to group together agents that have to work together. However, this proposal should not stay on static grounds, but propose a way to group together active agents that work together.
3. **Framework/component approach.** Modern software engineering has shown the importance of the framework/component concept. A framework is an abstract architecture in which components plug-in. It is often necessary to define sub-frameworks of frameworks. Unfortunately, in ACMAS, there is only one framework, the platform itself, and it is not possible to describe sub-framework in which specific interactions could be built.

To overcome these difficulties, Jennings proposes [9] a solution in the definition of a social level characterization of agent based systems, which follows Newell's levels of computer systems. However, this paper did not develop the main features of organization and their consequences in the process of analysis and design of MAS.

In the following, we will extend and continue these prospects by presenting and analyzing the main concepts of organization centered multi-agent systems (OCMAS) and their properties for building MAS. During our discussion, we will focus on a specific model of OCMAS, called AGR, for Agent/Group/Role, a simple though very powerful and generic organizational model of multi-agent systems.

3 Organization Centered MAS

3.1 Definitions

There are several definitions of what an organization exactly means. Indeed, the word "organization" is a complex word that has several meanings. In [6], Gasser proposed the definition of organization to which we subscribe:

An organization provides a framework for activity and interaction through the definition of roles, behavioral expectations and authority relationships (e. g. control).

This definition is rather general and does not provide any clue on *how* to design organizations. In [15] Jennings and Wooldridge propose a more practical definition:

We view an organisation as a collection of roles, that stand in certain relationships to one another, and that take part in systematic institutionalised patterns of interactions with other roles".

However, this definition lacks a very important feature of organizations: their partitioning, the way boundaries are placed between sub-organizations. Except in very small organizations, organizations are structured as aggregates of several partitions which may overlap, sometimes called groups or communities, contexts, department, services, etc. and each partition may itself be decomposed into sub-partitions. From these definitions, it is possible to derive the main features of organizations:

1. An organization is constituted of agents (individuals) that manifest a behavior.
2. The overall organization may be partitioned into partitions that may overlap (we will call these partition groups from now on)
3. Behaviors of agents are functionally related to the overall organization activity (concept of role).
4. Agents are engaged into dynamic relationship (also called patterns of activities [6]) which may be "typed" using a taxonomy of roles, tasks or protocols, thus describing a kind of supra-individuality.
5. Types of behaviors are related through relationships between roles, tasks and protocols.

An important element of organizations is the concept of *role*. A role is a description of an abstract behavior of agents. A role describes the constraints (obligations, requirements, skills) that an agent will have to satisfy to obtain a role, the benefits (abilities, authorization, profits) that an agent will receive in playing that role, and the responsibilities associated to that role. A role is also the placeholder for the description of patterns of interactions in which an agent playing that role will have to perform (in this paper, we do not distinguish between role and role assignment as in [12]). Organization may be seen at two different levels: at the organizational (or social) level and at the concrete (or agent) level (from [3]):

We will call *organizational structure* [11] (or simply structure, if there is no ambiguity) what persists when components or individuals enter or leave an organization, i.e. the relationships that makes an aggregate of elements a whole. Thus, the organizational structure is what characterizes a class of concrete organizations at the abstract or organizational level.

Conversely, a *concrete organization* (or simply organization), which resides at the agent level, is one possible instantiation of an organizational structure. This is a realization consisting of entities that effectively take part in a whole, together with all the links that bring these agents into association at any given moment. It is possible to relate an organizational structure to a concrete organization, but the same organizational structure can act as a basis for the definition of several concrete organizations

An organization consists in two aspects: a *structural aspect* (also called static aspect) and a *dynamic aspect*:

The structural aspect of an *organization* is made of two parts: a *partitioning structure* and a *role structure*. A partitioning structure indicates how agents are assembled into groups and how groups are related to each other. A role structure is defined, for each group, by a set of roles and their relationships. This structure defines also the set of constraints that agents should satisfy to play a specific role and the benefits resulting to that role. The *dynamic aspect* of an organization is related to the institutionalized patterns of interactions that are defined within roles. It defines also the modalities to create, kill, enter groups and play roles, how these modalities are applied and how obligations and permissions are controlled, how partitioning and role structures are related to agents' behaviors.

3.2 General Principles of OCMAS

Previous sections have allowed us to understand the basic concepts of organizations. It is now time to consider multi-agent systems from an organizational perspective. The question now is: what are the main principles from which organization centered multi-agent systems (OCMAS) may be approached for both analysis and design? The use of organizations provides a new way for describing the structures and the interactions that take place in MAS. The organizational level, the way organizations are described is thus situated in another level than the agent level that is often the only level considered in ACMAS. This level, which may be called "organizational level" (or "social level" as in [9]) is responsible for the description of the structural and dynamical aspects of organizations. This organizational level is an abstract represen-

tation of the concrete organization, i.e. a specification of the structural and dynamical aspects of a MAS, which describes the expected relationships and patterns of activity which should occur at the agent level and therefore the constraints and potentialities that constitute the horizon in which agents behave.

Principle 1: The organizational level describes the “what” and not the “how”. The organizational level imposes a structure into the pattern of agents’ activities, but does not describe how agents behave. In other terms, the organizational level does not contain any “code” which could be executed by agents, but provides specifications, using some kind of norms or laws, of the limits and expectations that are placed on the agents’ behavior.

Principle 2: No agent description and therefore no mental issues at the organizational level. The organizational level should not say anything about the way agents would interpret this level. Thus, reactive agents as well as intentional agents may act in an organization. In other words, ant colonies are as much organizations as human enterprises. Moreover, seen from a certain distance, or using an intentional stance [2] it is impossible to say if the ants or the humans are intentional or reactive. Thus, the organizational level should get rid of any mental issues such as beliefs, desires, intentions, goals, etc. and provide only descriptions of *expected* behaviors.

Principle 3: An organization provides a way for partitioning a system, each partition (or groups) constitutes a context of interaction for agents. Thus, a group is an organizational unit in which all members are able to interact freely. Agents belonging to a group may talk to one another, using the same language. Moreover, groups establish boundaries. Whereas the structure of a group A may be known by all agents belonging to A, it is hidden to all agents that do not belong to A. Thus, groups are opaque to each other and do not assume a general standardization of agent interaction and architecture.

These principles are not without consequences:

1. An organization may be seen as a kind of dynamic framework where agents are components. Entering a group/playing a role may be seen as a plug-in process where a component is integrated into a framework.
2. Designing systems at the organizational level may leave implementation issues, such as the choice of building the right agent to play a specific role, left opened.
3. It is possible to realize true “Open System” where agent’s architecture is left unspecified.
4. It is possible to build secure systems using groups as “black boxes” because what happens in a group cannot be seen from agents that do not belong to that group. It is also possible to define security policies to keep undesirable agents out of a group.

4 AGR: A Basic Model of OCMAS

In order to show how these principles may be actualized in a computational model, we will present the basics and methodology of the Agent/Group/Role model, or AGR

model for short, also known as the Aalaadin model [4] for historical reasons. We show that this model complies with the OCMAS general principles that we have proposed in the previous section.

4.1 Definitions and Axioms

The AGR model is based on three primitive concepts, Agent, Group and Role that are structurally connected and cannot be defined by other primitives. They satisfy a set of axioms that unite these concepts.

Agent: an agent is an active, communicating entity playing *roles* within *groups*. An agent may hold multiple roles, and may be member of several groups. An important characteristic of the AGR model, in accordance with the principle 2 above, is that no constraints are placed upon the architecture of an agent or about its mental capabilities. Thus, an agent may be as reactive as an ant, or as clever as a human.

Group: a group is a set of agents sharing some common characteristic. A group is used as a context for a pattern of activities, and is used for partitioning organizations. Following principle 3, two agents may communicate if and only if they belong to the same group, but an agent may belong to several groups. This feature will allow the definition of organizational structures.

Role: the role is the abstract representation of a functional position of an agent in a group. An agent must play a role in a group, but an agent may play several roles. Roles are local to groups, and a role must be requested by an agent. A role may be played by several agents.

We denote by $x.send(y,m)$ the action of an agent x sending a message m to an agent y , by $roleIn(r,g)$ the statement that the role is defined in a group g , and by $plays(a,r,g)$ the statement that the agent a plays the role r in g . We also denote by $GStruct(g,gs)$, the statement that g is a group considered as an instance of the group structure gs , and $member(x,g)$ the statement that an agent x is a member of a group g . Here are the axioms of the structural aspect of the AGR model:

a) Every agent is member of a (at least one) group:

$$\forall x:Agent, \exists g:Group, member(x,g)$$

b) Two agents may communicate only if they are members of the same group:

$$\forall x,y:Agent, \forall m:Message, x.send(y,m) \Rightarrow \exists g:Group, member(x,g) \wedge member(y,g)$$

c) Every agent plays (at least one) role in a group:

$$\forall x:Agent, \forall g:Group \Rightarrow \exists r:Role, plays(x,r,g)$$

d) An agent is a member of the group in which it plays a role:

$$\forall x:Agent, \forall g:Group, \forall r:Role \\ plays(x,r,g) \Rightarrow member(x, g)$$

e) A role is defined in a group structure:

$$\forall x:\text{Agent}, \forall g:\text{Group}, \forall r:\text{Role}, \text{plays}(x,r,g) \Rightarrow \exists gs:\text{GroupStructure} \wedge \text{GStruct}(g,GS) \wedge \text{roleIn}(r,GS)$$

Roles may be described as in Gaña [15] by attributes such as its cardinality (how many agents may play that role). It is also possible to describe *structural constraints* between roles. A structural constraint describes a relationship between roles that are defined at the organizational level and are imposed to all agents. In AGR, we propose two structural constraints: *correspondence* and *dependence*. A correspondence constraint states that agents playing one role will automatically plays another role. For instance, to express the, quite classical, political correspondence between delegates of smaller groups (states, departments, regions) which are automatically members of another group where they act as representative (deputies, ambassador, etc.) we would use the following statement:

$$\text{Role}(\text{'delegate'},GS1) \rightarrow \text{Role}(\text{'representative'},GS2)$$

where GS1 and GS2 are group structures. This constraint may be defined as follows:

$$\forall x:\text{Agent}, \forall g:\text{Group}, \text{where } \text{GroupStructure}(g,GS1), \exists g':\text{Group} \text{ where } \text{GroupStructure}(g',GS2) \text{ such that: } \text{plays}(x,\text{'delegate'},g) \Rightarrow \text{plays}(a,\text{'representative'},g')$$

If the two roles have the same set of members, we will use the notation \leftrightarrow . For instance, in most human organizations (associations, corporation, syndicates, etc.), all voters are eligible. In our notation, we would express this constraint as:

$$\text{role}(\text{'voter'},GS1) \leftrightarrow \text{role}(\text{'eligible'},GS1)$$

whose definition is as follows:

$$\forall x:\text{Agent}, \forall g:\text{Group}, \text{where } \text{GroupStructure}(g,GS1), \text{plays}(x,\text{'voter'},g) \Rightarrow \text{plays}(x,\text{'eligible'},g) \wedge \text{plays}(x,\text{'eligible'},g) \Rightarrow \text{plays}(x,\text{'voter'},g)$$

Dependence constraints express dependencies between group membership and role-playing. For instance, an agent is authorized to be a director of a Laboratory only if it is also a researcher in the lab. This would be expressed in the following way:

$$\text{Role}(\text{'director'},\text{'Lab'}) \text{ requires } \text{Role}(\text{'researcher'},\text{'Lab'})$$

Its semantics could be defined in a 1st order logic as follows:

$$\forall x:\text{Agent}, \forall g:\text{Group}, \text{where } \text{GroupStructure}(g,\text{'Lab'}), \text{plays}(x,\text{'director'},g) \Rightarrow \text{plays}(a,\text{'researcher'},g')$$

The AGR meta-model is represented figure 1 in UML. Several notations may be used to represent organizations. In [13] a notation based on UML has been proposed to represent groups and roles. This is a very convenient notation to represent the abstract structures of an organization, but concrete organizations cannot be represented in this notation. This is why we will use the following another notation, that we call the *cheeseboard diagram*, which is very convenient to represent examples of concrete organizations.

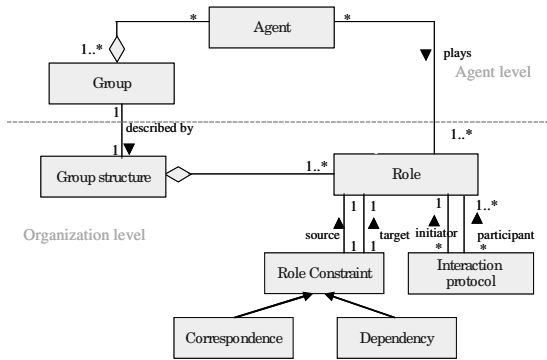


Fig. 1. The UML meta-model of AGR

4.2 The “Cheeseboard” Diagram

In the cheeseboard diagram, a group is represented as an oval that looks like a board. Agents are represented as skittles that stands on the board and sometimes go through the board when they belong to several groups.

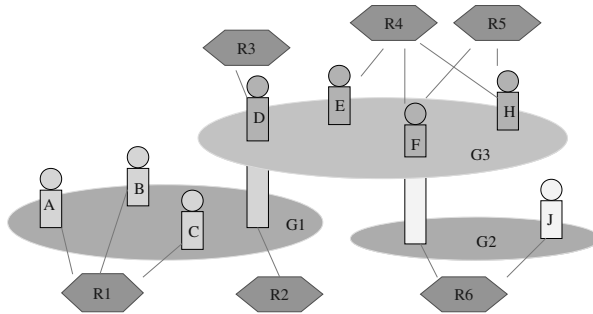


Fig. 2. The "cheeseboard" notation for describing concrete organizations

A role is represented as a hexagon and a line links this hexagon to agents. Figure 2 gives an example of a concrete organization using the cheeseboard diagram. In this picture, the agent F is a member of both G2 and G3, playing roles R4 and R5 in G2, and R6 in G3.

4.3 Describing Organizational Structures

The cheeseboard notation, while very suitable for concrete organization, is not suited to the description of relationships within organization at an abstract level, i.e. for the definition of organizational structures. Thus, we have introduced a notation for describing organizational structures.



In order to express organizational diagrams in a more simple and convenient way, we propose a set of graphical items. In this notation, *group structures*, i.e. abstract representation of groups, are represented as rectangles in which roles, represented as hexagons, are located. Constraints are represented as arrows between roles. We use two kinds of arrows. Large arrows are used for correspondence and thin arrows are used for modeling dependencies.

Interaction diagrams, which are represented as rounded rectangles, are used to describe communication protocols between roles. Without considering the way agents communicate, it is possible to describe communications at an abstract level, i.e. as specific constraints between roles. An interaction may take place between two or more agents and is described at the organizational level between roles. The role initiator of the interaction is represented by an arrow that points towards the interaction. Other participating roles are represented as simple lines between interaction and roles. The figure 3 shows an organizational structure related to the concrete organization of figure 1. In this diagram, many different cases are represented. There are 3 group structures, called GS1, GS2 and GS3. The dependency d1 expresses a correspondence between the role R2 of GS1 and the role R3 of GS2. This allows for the definition of agents that act as representative between two groups. The dependency d2 expresses a dependency between R4 and R5, which means that all agents playing R5 must play R4. Interactions I2, I5 and I6, which are related to only one role, will be performed by different agents playing the same role. The interaction I3 takes place between agents playing three roles. Interactions may be figured by different types of diagrams: automata, Petri nets or sequence organizational diagrams.

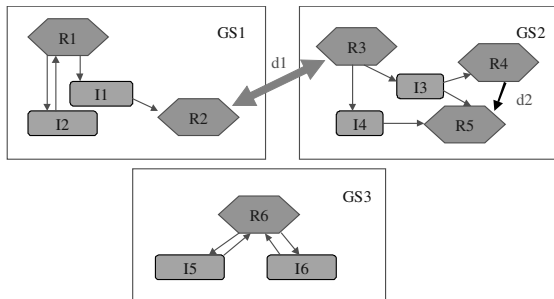


Fig. 3. Organizational structure representation

4.4 Describing Organizational Activities

To describe the dynamics of organizations, i.e. the temporal relation that is expressed between organizational events, such as the creation of groups, the entering or leaving of a group by an agent or the acquisition of a role in relation, we will use a specific notation, that we call *organizational sequence diagram*, which is a variant of the sequence diagram of UML (or AUML) [1].

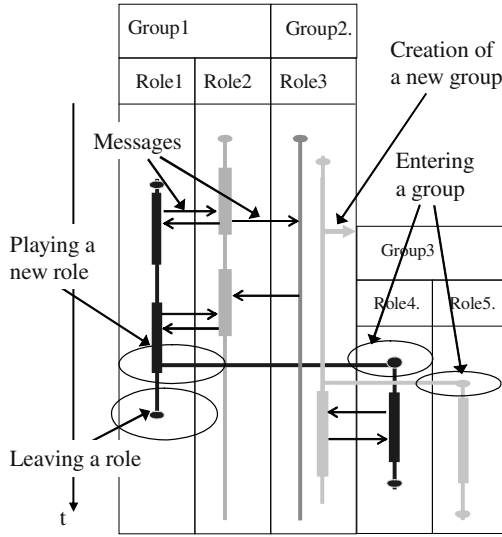


Fig. 4. The organizational sequence diagram

Whereas in AUML vertical lines correspond to agents, in our diagram, the life of an agent is made of several segments of the same color (unfortunately, colors are displayed as gray levels in this paper). Each segment describes the “life” of an agent playing a specific role in a specific group. Thus, it is possible to represent the fact that an agent may belong to several groups and play several roles at once. Figure 4 shows a general view of this type of diagram.

4.5 Groups Dynamics

Groups may be built at will. A group is created upon request of an agent, from an already described group structure. A group structure may be ‘blank’, thus allowing agents to build roles at will and to enter groups without any limits. However, in the general case, entering a group is a rather complex process, because an agent has to be authorized to enter a group. Due to axiom b) an agent cannot communicate directly to agents belonging to the group. Thus, it cannot request a permission to enter a group to agents belonging only to that group. A solution to this problem lies in the organization itself, in its possibility to build complex organizational structures. We will assume that an agent is permitted to enter a group only if it provides the right authorization. This agent could get this authorization in an “examination” like organizational pattern. An ‘entrance’ group, associated to the group A, acts as an “air lock” between the group A and its exterior. There is no authorization required for A to get the ‘candidate’ role in an entrance group. The ‘gatekeeper’ agent could then check the conformity of this agent to the specification of the structure and roles of the group A. Figure 5 shows this adhesion process using a cheeseboard diagram. The semantics of this process has been described in [5] using a variant of the π -calculus.

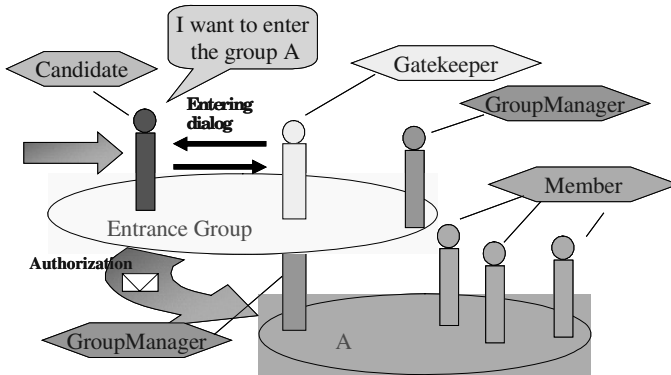


Fig. 5. The cheeseboard representation of a group adhesion process.

It should be clear that this is only a simple aspect of all the organizational patterns that could be used to manage the organizational activities of an OCMAS. We just wanted to show that *it is possible to manage an OCMAS using only OCMAS features!*

Obviously, when it will come to implementation of agents, designers would have to relate the architecture and the cognitive properties of their agents to the organizational structure and dynamics of such a system. We only claim that, in OCMAS, this aspect would be considered in a second phase.

5 Methodology

Notations are not sufficient to describe a methodology. In this paper, we will only briefly suggest the key point of how a methodology could be defined on an OCMAS model. The designer should first identify the main groups of the application. A group may be used for two main purposes:

- **To represent a set of similar agents.** In this case, a group is merely a collection of agents that exhibit certain similarities. There are usually few roles and a role may contain many agents. For instance, in AGR, to have a set of agents using the same communication language, such as ACL FIPA, one could design a FIPA group. Then the FIPA agents called the Directory Facilitator (DF) and the Message Transport Service (MTS) would be represented as agents playing the DF and MTS roles respectively. All other agents would merely have a simple 'member' role.
- **To represent a function based system:** each role then corresponds to a function or a subsystem of a whole system. Agents then act as specialists characterized by their skills to achieve functions associated to the roles. For instance in a computer network, printers have the ability to print and may be associated to the role of 'printer'. A soccer robot team would have the roles 'goalkeeper', 'leader', 'attacker', 'middle', etc.

Once these groups have been identified, the overall organizational structure is built using some organizational patterns [7, 11] such as the e-commerce organizational pattern that is presented in the next section as an example.

The partitioning of agents describes the way an organization is decomposed into its sub-components, and optionally the way these sub-components are further decomposed into their own sub-components, and the way these sub-components are aggregated. In AGR, hierarchies of groups, also called holarchy by Odell and Parunak [13] where a group is represented by an agent at the next level, may be represented by an organizational pattern where some ‘delegate’ agents in one group are seen as ‘representative’ agents in another group.

When the organizational structure is built together with organizational dynamic of group creation and adhesion, it is time to get into the definition of roles in a functional way. Then one could use the Gaia [15] methodology to fill the roles and relate them to the general structure. Our vision has some connection with object-oriented design, where the key diagrams are the class diagrams, which represent the static aspects of objects, and the sequence diagrams, which gives an insight of the dynamic aspects of objects. We use the same kind of distinction with the organizational structure diagrams and the organizational sequence diagrams. However, we often use the cheeseboard diagram to get a first idea of the organizational patterns one could use to build an OCMAS.

6 Example

In order to explain how the AGR model may be used for analyzing and designing multi-agent systems, we will present an example taken from a situation that all researchers know very well: the “reviewing process” of papers in a conference. There are three group structures: the program committee group structure, the submission group structure (for a given conference there is only one group for each of these group structures), and the evaluation group structure. The program committee has only two roles: a program chair and a PC member. The submission group contains also two roles: submission receiver, which receives papers, and author. There are several submission groups, and the reviewing manager must be part of the program committee group. It is clear from this diagram that agents may belong to different groups: a committee member may be a reviewing manager of an evaluation group *and* an author submitting a paper (Fig. 6).

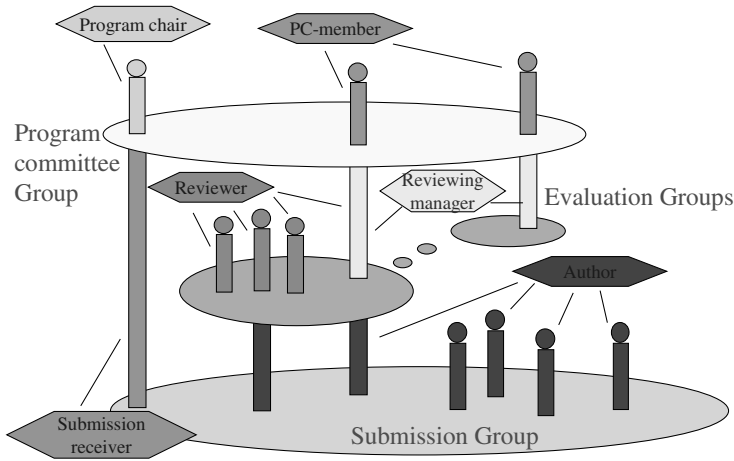


Fig. 6. The organization of a program committee using a cheese-board notation

The figure 7 presents the organizational structure of such an organization. Interactions such as ‘distribute papers to review’ or ‘notification of acceptance’ are protocols that relate agents through their roles. These protocols could be represented by any kind of diagram for representing protocols (finite state automata, Petri nets, etc.).

Figure 8, shows an organizational sequence diagram representing some part of the reviewing process. An author submits a paper to the submission receiver which is also the program chairman. As such, he/she asks a program committee member to review the paper. Then, this member creates a submission group and distributes the paper to the reviewers. When the reviewer has done its job, the committee member says that the paper is accepted (or rejected) and the submission receiver then sends an acceptance message to the author. Doing so, the author is therefore accepted as a speaker of the conference.

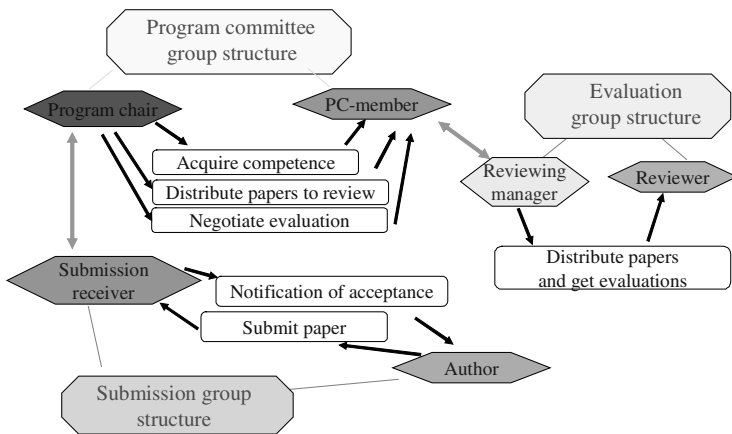


Fig. 7. The organizational structure diagram of the reviewing process example.

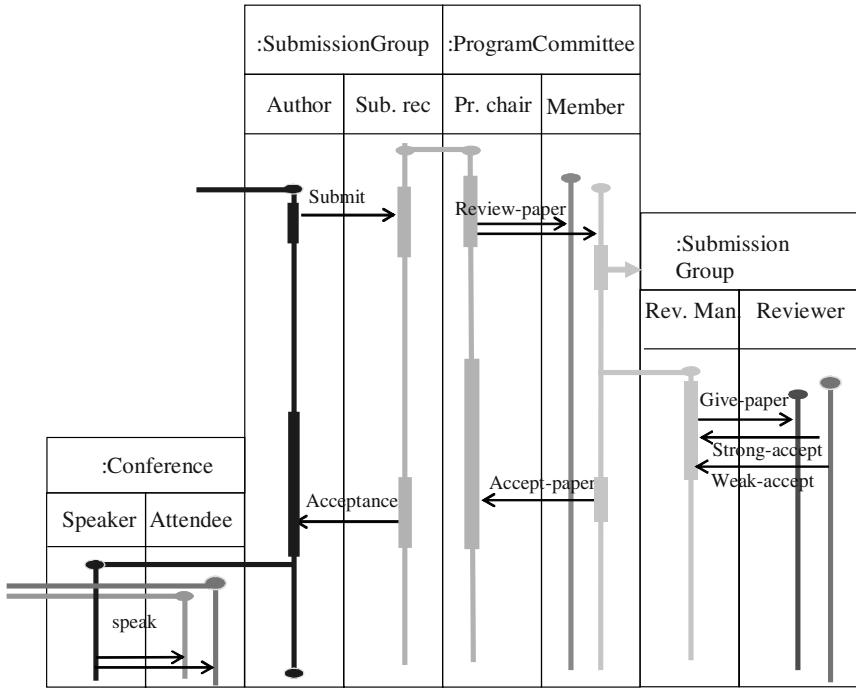


Fig. 8. A possible organizational sequence diagram of the reviewing process organization.

7 Conclusion

In the introduction, we have claimed that ACMA have some drawbacks that OCMAS may resolve. For this reason, we have proposed a general framework to understand and design MAS based on organizational concepts such as groups, roles and interactions, which may overcome some of the weaknesses of ACMA. We have presented the AGR model in this framework, showing how it is possible to design applications using these concepts that totally adhere to the OCMAS principles that we have introduced previously (see section 3.2):

1. The AGR architecture does not describe the “how”, and only specifies the “what” by describing organizational structures made of group structures and roles.
2. We have not used mental issues such as intentions, goals or beliefs to describe the AGR model. We do not say that we should not use them: only that it is possible to build complex MAS architecture without using them. It is then the responsibility of the design process to describe agents able to live and interact in such architectures.
3. AGR provides a way for partitioning a system through the concept of group.

Thus, the main drawbacks of ACMAS disappear: it is possible to build secure applications at the group level, by designing gate keeper roles that prevent unauthorized agents to enter a group, or by describing norms (obligations, permissions, interdictions) that are related to groups and roles (this latter feature will be presented in a forthcoming paper). Complex programs may be built by using groups as dynamic frameworks that agents may create, enter and leave at will during their lifetime. In software engineering terms, agents may now be considered as some kind of dynamic “components” that live in dynamic frameworks.

Moreover, we claim that AGR is certainly one of the smallest possible organizational models. The structure or roles, is left open for the moment, but may be extended by integrating the most recent propositions on the nature of roles (see for instance [12]).

We have presented a set of diagrams (organizational structure, “cheeseboard” diagram, and organizational sequence diagrams) which may represent the different aspects of OCMAS. We have also sketched how these concepts may be used in a methodology based on organizational principles.

Organizational concepts may be used for practical implementations. The MadKit platform [8] that we have designed is built around the AGR model. Since its first release, hundreds of users (thousands of downloads) have been able to use these organizational concepts (presented in a less rigorous way than here) to build applications in various areas.

Many aspects of organizations, such as functional views, deontic aspects (concepts of norms and institutions) and the use of reflection to build complex MAS platform have been left over and will be presented in future papers.

References

1. Bauer, B., Müller, J.P. and Odell, J., Agent UML: A Formalism for Specifying Multiagent Interaction. in *Agent-Oriented Software Engineering*, (2001), Springer, 91–103.
2. Dennett, D.C. *The Intentional Stance*. M.I.T. Press, Cambridge, Massachusetts, 1987.
3. Ferber, J. *Multi-Agent Systems: an introduction to distributed artificial intelligence*. Addison-Wesley, 1999.
4. Ferber, J. and Gutknecht, O., Aalaadin: a meta-model for the analysis and design of organizations in multi-agent systems. in *Third International Conference on Multi-Agent Systems*, (Paris, 1998), IEEE, 128–135.
5. Ferber, J. and Gutknecht, O., Operational Semantics of a Role-Based Agent Architecture. in *Agent Theories, Architectures and Languages*, (Orlando, 2000), Springer-Verlag.
6. Gasser, L. An Overview of DAI. in Gasser, L. and Avouris, N.M. eds. *Distributed Artificial Intelligence: Theory and Praxis*, Kluwer Academic Publishers, 1992, 9–30.
7. Giorgini, P., Kolp, M. and Mylopoulos, J., Organizational Patterns for Early Requirement Analysis. in *IEEE Joint Int. Requirements Engineering Conference (RE'02)*, (Essen (Germany), 2002).
8. Gutknecht, O., Michel, F. and Ferber, J., Integrating Tools and Infrastructure for Generic Multi-Agent Systems. in *Autonomous Agents 2001*, (Boston, 2001), ACM Press, 441–448.
9. Jennings, N.R. On Agent-Based Software Engineering. *Artificial Intelligence*, 117 (2), 277–296.

10. Jennings, N.R. and Wooldridge, M. Agent-Oriented Software Engineering. in Bradshaw, J. ed. Handbook of Agent Technology, AAAI/MIT Press, 2000.
11. Mintzberg, H. The Structuring of Organizations. Prentice-Hall, 1979.
12. Odell, J. and Parunak, H.V.D., The Role of Roles in Designing Effective Agent Organizations. in Software Engineering for Large-Scale Multi-Agent Systems, (2003), Springer.
13. Parunak, H.V.D. and Odell, J., Representing Social Structure in UML. in Agent-Oriented Software Engineering II, (Montreal Canada, 2002), Springer, 1–16.
14. Rocha Costa, C. and Demazeau, Y., Toward a Formal Model of Multi-Agent Systems with Dynamic Organizations. in ICMAS'96, (Kyoto, 1996), AAAI Press.
15. Wooldridge, M., Jennings, N.R. and David, K. The Gaia Methodology for Agent-Oriented Analysis and Design. Journal of Autonomous Agents and Multi-Agent Systems, 3 (3). 285–312.
16. Zambonelli, F. and Parunak, H.V.D., From Design to Intentions: Signs of a Revolution. in AAMAS 2002, (Bologne (Italy), 2002), ACM Press, 455–456.
17. The Jade Platform : <http://sharon.csel.it/projects/jade>
18. The Fipa-OS platform: <http://sourceforge.net/projects/fipa-os>

Modelling Multi-agent Systems with Soft Genes, Roles, and Agents*

Qi Yan, XinJun Mao, Hong Zhu, and ZhiChang Qi

Department of Computer Science and Technology, Team 6,
National University of Defense Technology,
Changsha, 410073 China, +86 731 4535411
yanqi_nudt@yahoo.com.cn

Abstract. The underlying homogeneousness between human being in a society and software agent in a MAS (Multi-Agent System) is opened out in this paper, based on which a new definition and architecture for agent is proposed. An agent is made of certain soft genes and roles. Such agents can be more appropriate for dynamic, open system's analysis, design and implementation. The associated structure model and behavior model of MAS are described. A MAS modelling methodology RoMAS (Role-based MAS modelling methodology) based on these definitions is presented through RoboCup simulation football team case, including its graphical modelling language and the modelling process.

1 Introduction

Agent-Oriented Software Engineering (AOSE), Multi-Agent System (MAS) [1, 2] and Agent-Oriented Programming (AOP) [3,4] attract much attentions in software development field. They are even regarded as a revolution by some researchers [5]. Meanwhile, Agent-Based Social Simulation (ABSS) receives social researchers' great attentions, e.g. RoboCup [6] has already become an ideal test-bed for both ABSS and MAS.

Since it is agent that associates ABSS and MAS, the first question we need to answer is "what are agents?". This paper will address this question in a new way and propose a MAS modelling methodology to make Multi-Agent Based Simulation (MABS).

The remainder of this paper is organized as follows. Section 2 illustrates the homogeneousness between human being and software agent. Section 3 presents the main idea of soft gene, role and agent. Section 4 introduces agents structure and behavior model. Section 5 illustrates the usefulness of the concepts by modelling a RoboCup [6] simulation football team using RoMAS (Role based MAS modelling methodology) [7]. Section 6 discusses related works. Section 7 gives some conclusions and discusses the future works.

* This work is supported by the National Science Foundation of China under Grant No.600003002; the National High Technology Development 863 Program of China under Grant No.2002AA116070.

2 From Human Study to MABS

From Descartes, Kant, Floyd, etc. to Karl Marx, after their studying of nearly 400 years, human essence research recognized that human individual holds both natural and social properties [8]. Such a theory may help software researchers to understand agent and multi-agent system for that human and society give good examples for agent and MAS, from basic apperceive, interaction to challengeable dynamic, open properties.

2.1 Traditional Agents: Make Actions and Interactions

Current agent-oriented methodologies (see [9] for a survey of them) lack of the ability to deal with distributed systems' open, dynamic properties for which the radical reason is the understanding and definition of *agent*.

Existing agent-oriented software methodologies, e.g. Gaia [4], usually take an agent as *a software component that autonomously behaves and collaborates with others in certain environments*. But how to add or change interaction of new types between agents at run-time? These problems are cause by open and dynamic properties of MAS and have not been addressed by the methodologies [10]. Currently, some agent-oriented software methodologies introduce multiple inheritance and dynamic inheritance mechanisms to deal with the problems, while they are recognized as negative factors for system robustness and maintainability.

To address these problems, we may need a new understanding about agent. We notice that an agent is quite like a human individual in our society — it can make actions and interact with others, which gives us big intellectual enlightenment.

2.2 Human Individuals Have Genes and Roles

Karl Marx said in *On Feuerbach, 1845*, "The first premise of all human history is, of course, the existence of living human individuals. Thus the first fact to be established is the physical organization of these individuals and their consequent relation to the rest of nature."

As Karl Marx mentioned, 'the existence of living human individuals' and 'the physical organization of these individuals' are essential to any intelligent individual. Our understanding is that two concepts are essential to creatures: gene and role. A gene is a hereditary unit that occupies a specific location on a chromosome and determines a particular characteristic in an organism. Genes exist in a number of different forms and can undergo mutation [11]. Dawkins said in his famous book *The SELFISH GENE* [12] that human beings are nothing but gene machines. We accept his theory and regard human beings behave under the control his genes. Later we will illustrate how to distill basic behaviors and characters from software agents and then get *soft genes*. A role is the characteristic and expected social behavior of an individual [11]. Roles make human beings have social intelligence of communicating and cooperating.

2.3 New Agents: Learn from Human through Homogeneousness

To understand human beings, several questions need to be answered. They are described in figure 1. It represents:

1. What activities do human beings do? Human beings do two typical kinds of activities (noted as rectangles): (1) Practice Activity: A human lives in the world, he/she can apperceive from/react to the world, E.g. when a football player kicks a football without any intention, he/she is doing a practice activity. Such an activity may be the reaction to environment's alternation, instinctive actions under genes' control, etc.; (2) Social Interaction Activity: A human lives in a society, he/she takes certain social roles so as to interact with others, E.g. when a football player (as a vanguard) transports a football to his/her teammates (maybe an attacker), a social interaction activity is happening.
2. What medias do human beings use? Human beings use two typical kinds of medias (noted as rectangles):(1) Tool System: Human beings use tool system to perform practice activities (as the dashed line describes), e.g. eyes, ears, mouth, hammer, etc.; (2) Social Interaction System: Human beings use social interaction systems to perform social interaction activities. These interaction systems include English grammar, contract, etc.
3. What relations do human beings have? Three circles respectively represent three layers in which an agent reflect to itself or relate to other agents or the environment. The *Being* is the human self, which can be seen as the container of genes and roles. Before it is combined with some *Individual's* mental attitudes and to *Role organization*, it can not do anything. In *Individual* layer, it relates to itself mental attitudes and it makes practice activities (as the dashed line describes); in *Role organization* layer, it relates to other agents through their roles and it makes social interaction activities; in *Environment* layer, it relates to real/virtual world objects and it makes practice activities.

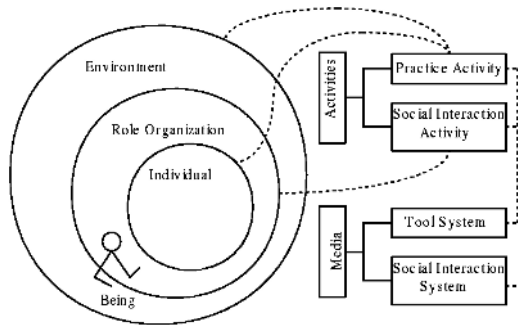


Fig. 1. Human being's activities, medias and layers

When we homogeneously map human properties to agent (although we have not defined agent yet, we still can use this word to see what properties it should have), we get figure 2. It represents:

1. What activities does an agent do? An agent does two typical kinds of activities (noted as rectangles, the curved lines connecting circle/ellipse side and rectangles represent an association relation):(1)Practice Activity (arrows between agent and environment/itself): An agent lives in the software system, it should be able to apperceive from/react to the system, E.g. when a software football player kicks a football, it is doing a practice activity. Why and how such activities be performed? Such an activity should be under something's control. We call it soft genes (noted as SGi in figure 2, i is 1,2,3,etc.) which is defined in the next subsection; (2)Social Interaction Activity (arrows between agents represent social interaction activities, the curved lines connecting arrows and rectangles represent an association relation): An agent lives in a virtual society, it takes certain social roles so as to interact with others, E.g. when a software football player (as a vanguard) transports a football to its teammates (maybe an attacker), a social interaction activity is happening.
2. What models does an agent use to make activities? Agents use these typical kinds of models (noted as rectangles, the curved lines connecting circle/ellipse side and rectangles represent an association relation, the curved dashed lines represent some activity use some models):(1)Model on itself: Agents use Function Libs, Behavior Rules, Role Alternation Rules (see section 4) to determine what to do at certain time; (2)Role Organization Model and Communication Protocol: Agents use these models to perform social interaction activities. Notice that we regard environment model belongs to (2) since the environment can also be a role.
3. What relations does an agent have? In 'SGi' layer (smaller ellipses), it relates to itself and it makes practice activities; in 'Role organization' layer (bigger ellipses), it relates to other agents through their roles and it makes social interaction activities; in 'Environment' layer (circles), it relates to virtual world objects and it makes practice activities.

Now that we have already completed the homogeneousness analysis, it is necessary for us to distill substaintial concepts from the complex phenomenon as figure 2 shows so as to make it easy for understanding them.

2.4 A New Understanding for Agents

Thinking from a bionic point of view, software agents bear an analogy to human beings for that they both require the properties of autonomy, reactive, etc. We propose two concepts, i.e., soft gene and role, for further defining software agent.

1. A soft gene (for agent) is a hereditary unit that determines some particular characteristics in a software component. It defines certain physical attributes

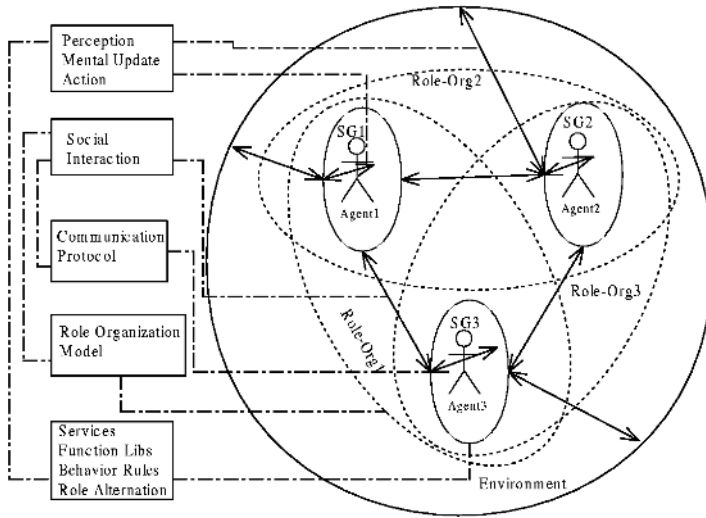


Fig. 2. Agent's activities, models and layers

and behaviors. After adopting certain soft genes, the software component (agent) can make certain practice activities, update its own behavior rules, etc. In other words, a software component with certain soft genes can do something and knows how to do them by itself.

2. A role (for agent) is the characteristic and expected social behavior of a software component. It defines certain responsibilities and cooperation patterns. After binding certain roles, the software component (agent) can know and make social interaction activities. In other words, a software component with certain roles can do something and knows how/why to do them with others.

An agent is a software component that composed by certain soft genes and bound with certain roles.

In the next section, we will give formal definition for these concepts. Here, we regard *environment* as a role which may seldom affected by other system roles or may be a quite active role in an open and dynamic system, then the third layer agents live in can be reduced to role layer and soft gene layer. The importance of such a reduction is that we can use as least concepts as possible to simulate a society with MAS. We may simulate *environment* of an agent as all the other agents. However, since it does not affect what we discuss about soft genes and roles, we leave it for further research.

3 Soft Gene, Role, and Agent

The meta-model of multi-agent system is as below: an agent is composed by a certain set of soft genes bound with a certain set of roles; an agent's ability of

action is determined by its genes; an agent communicates with others through its bound roles.

3.1 Soft Gene

We have defined a soft gene is a hereditary unit that determines some particular characteristics in a software component. For example, a football (an object in a computer simulation football match) is a software component and its soft genes are *shape*, *color*, *texture*, *springiness*, *size*, *weight* and so on. We may define a component with such genes:

< Genes — — *shape* : *round*; *color* : *black and white*; *size* : *24cm*; *texture* :
leather; *weight* : *430g*; *springiness* : *good* >

It can almost be assured that we are talking about a football. However, soft genes are not just attributes. Considering another software component in a simulation football match, a player, his/her soft genes are both attributes (*height*, *weight*, *run speed* and so on) and behaviors (*run()*, *goal()*, *grab()* and so on) with the ways in that those behaviors perform (some player like to give the football to teammates while some others will take the football by themselves).

Once the soft genes are determined, the software component's *existence of living ... individuals* is completely described. (*The physical organization of these individuals* will be discussed in the next subsection.) Before we can move to definition of any of above concepts, we must first define some primitives. A behavior is an action's being called from outside/self and performed to cause a discrete event that changes the state of the environment. An attribute is a perceivable feature (of the world).

Definition 1. *A soft gene is an entity that comprises a set of behaviors and a set of attributes.*

In Z [13], before constructing a specification, we must first define types. Here we define the set of all behaviors and the set of all attributes:

[*Behavior*, *Attribute*]

We note that behavior and attribute are considered as basic types and no further definitions are given. Now a state schema can be constructed that defines a soft gene. (For the meanings of the Z notations, readers are referred [13].) In the schema, *e_cando*, *i_cando* are the sets of behaviors(external observable or internal observable) of the soft gene; *e_attributes*, *i_attributes* are the sets of features(external perceivable or internal perceivable) of the soft gene. Soft genes are therefore defined by their ability in terms of their behaviors, and their configuration in terms of their attributes.

Softgene

e_cando : P Behavior

i_cando : P Behavior

e_attributes : P Attribute

i_attributes : P Attribute

Some additional properties of soft gene should be illustrated:

1. Heteromorphism: soft genes can be organized into different abstract levels.
 - a) As software production: the genes are 1 and 0. Every software production is a sequence of 0 and 1;
 - b) As model: the genes are chosen characteristics of modelling entities. For example, when we simulate a football player, we just need to consider his/her weight, height, characteristics, ability of kicking not painting;
 - c) In implementation: the genes are certain programming language mechanisms, such as *Class*, *Interface* in C++, Java and so on.
2. Inhomogeneity: different applications require different soft genes. To simulate an office system, we care about officers' name, salary, ability to use computer, etc.; to simulate a football player, we care about his/her weight, height, characteristics, ability of kicking not painting.
3. Inheritance: genes can inherits from other genes. It is important for reuse.
4. Aggregation: genes can be aggregated into a bigger gene.

3.2 Role and Role Organization

Soft genes determine software individuals' existence of living, *The physical organization of these individuals* makes the backbone of one system. A role represents the expected social behavior of an individual. All roles as a whole can be regarded as the physical organization of a multi-agent system. For the last example, if a player has such role description:

$\langle \text{Roles} : - - \text{Always} : \text{Left guard}; \text{Possibly} : \text{Left front} \rangle$

We can state that this player should defense attacks from left side; most time he/she communicates with the goal keeper role.

Definition 2. *A goal is a state of affairs to be achieved in the environment.*

Definition 3. *A service is a behavior framework, which determines how the behavior be performed by the soft gene that binds the role.*

Goal == P Attribute
Service == P Behavior

Definition 4. *A role is an encapsulation of some attributes with the addition of goals and services.*

Role

services : P Service
goals : P Goal
e_attributes : P Attribute
i_attributes : P Attribute

goals $\neq \phi$
services $\neq \phi$

Some relations between roles are:

1. Relational: In any system, a role almost always appears with at least another role that it interacts with. E.g., whenever there is a role of teacher, there must be a role of student. Otherwise, the role of teacher is meaningless. In summary, a role defines the relationships among individuals, also the interface, protocol and functionality of the interactions among individuals.
2. Hierarchical: Those individuals who play a role in an organization can be highly organized. E.g., the lecturers in a university's department can be further divided into subject groups and play roles like lecturer in computer science and lecturer of information systems, etc. In this way, the duty and tasks of a role at a higher level are fulfilled through interactions between roles at a lower level. Therefore, a role organization can be decomposed hierarchically.

The representation of the structure of a dynamic system in the form of a role organization is usually stable in the sense that the relationships between the roles do not change so frequently as the those between the individuals of the system. However, the relationships between the roles are not always constant. In human organizations, new role can be formed and existing roles can be deleted or merged into another role, etc. The revision of the role organization by the organization itself leads to self-organized.

Roles together with the interaction paths among them compose a role organization. Many of role organizations are documented patterns (see Figure 3). Role organization is helpful in:

1. It represents how agents interact. Each agent acts and communicates under its role or roles.
2. As a frame in which agents bind certain roles, it enables the agents to change their roles dynamically.
3. Essentially, roles can be organized into another role [14]. It can be instantiated, generalized, specialized, and aggregated.

Considering a football team: the typical roles (goalkeeper, linebacker, vanguard and attacker) compose a role organization (Figure 3). Note that except goal-keeper, specializations of each role (e.g. left-backer, right-backer and middle-backer as specializations of linebacker) compose role organizations as well.

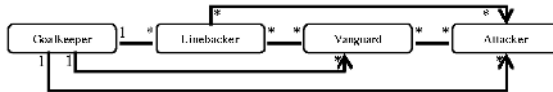


Fig. 3. The pattern of football team roles

The rectangle in figure 3 denotes role; the links represent communication paths; and notation 1 or * near link ends denotes quantities of the role.

In comparison with existing works in which role and role organization are utilized in analysis and design, we believe that as facilities for MAS modelling, they should also be applied in implementation process. The reason will be illuminated in section 6.

3.3 Agent

Each agent holds some soft genes and takes on one or more roles. Then an agent’s composition can be written as:

$$Agent = \langle \{Soft\ Gene\}, \{Role\} \rangle$$

Definition 5. *An agent is an entity that comprises some soft genes together with some bound roles.*

The relation between soft genes and roles is *maybound*, and the *bind* method is used to make such a relation between certain soft genes and roles.

$$| \text{ maybound} : Softgene \leftrightarrow Role$$

<p><i>Agent</i></p> <hr/> <p><i>softgenes</i> : P <i>Softgene</i> <i>roles</i> : P <i>Role</i> <i>bind</i> : <i>Role</i> → <i>Softgege</i></p> <hr/> <p><i>softgenes</i> ≠ ϕ <i>roles</i> ≠ ϕ <i>bind</i> ⊆ <i>maybound</i></p>

Because of limited space, we omitted other important schemas, e.g. agent state, agent act, etc.

3.4 Be Intelligent to Live in Open and Dynamic Systems

In an open and dynamic system, agents need to know what/how they can do by itself and what/how they can do with others. So long as an agent can alter its bound roles, we say it has the intelligence to interact with others under an open and dynamic environment. We believe that this definition and understanding of agent has such advantages to others:

1. The set of soft genes in an application is quite stable. Every individual (without interactions with others) is composed by some genes.
2. An agent can bind or discard some roles at run-time. This character is particularly useful for open and dynamic systems:
 - a) To create roles of some new types, the system just reads new role specifications and create corresponding roles.



- b) After discarding some old roles, the system organizes interactions among remaining agents through their bound roles.
 - c) By binding or discarding roles, an agent may dynamically alter its way of interaction with other agents.
3. For implementation, multiple inheritance will be not necessary and then corresponding side effects can be prevented.

As we have discussed in section 2, other agent definitions and methodologies can not address these problems well, such as Gaia [4] or MaSE [15,10].

4 Structure and Behavior Model of MAS

To use definitions we have given (soft gene, role, agent) to model a MAS, both structure and behavior models should be built.

4.1 Structure Model

The structure model includes:

1. Goal structure: The goal decomposition structure of the requirement of the MAS.
2. Role model: The inheritance and aggregation relations among roles. Besides, role model describes what goals and services included in the role.
3. Soft gene model: The inheritance and aggregation relations among soft genes. Besides, soft gene model describes what services supported by it.
4. Agent model: Describes how and what soft genes and roles combine to generate agents. The inheritance and aggregation relations among agents make up the structure model of agents.

When agent a_1 inherits from agent $a_2, a_3, etc.$ a_1 will inherit these agents' roles and visible genes, including all the attribute types and methods. When agent a_1 is aggregated by agent $a_2, a_3, etc.$ a_1 will get all these agents' roles and genes, whether they are visible or not. All the inheritance and aggregation relations among agents make the complete structure agent models. However, this is only simple introduction for inheritance and aggregation. In fact, we should write detailed axiomatic semantics for these two relations. The work is left for future works.

Agent structure model and role structure model together make up the system structure model.

4.2 Behavior Model

The behavior model includes:

1. Soft gene's behavior model: The methods of a soft gene stand for the gene's behaviors. Any behavior should be restricted by some behavior-rules, which are represented as pre-conditions and post-conditions in logic formulae.

2. Agent's role transition model: An agent may bind or discard its roles at run-time. The role transition can be modelled as a finite states machine (automata), in which from the states system can know the roles bound to the agent at that time; the transitions among states happen under certain conditions.
3. Role's interaction model: Sequence diagram may model roles' interactions well.

However, other models should be considered in modelling pragmatic systems. For example, use case diagram [16] is one important modelling language. We will represent a full version of models in section 5.

5 RoMAS: Role-Based MAS Modelling Methodology

Based on the definitions we have given, we propose a role-based modelling language and methodology tailored to (1) explicitly separate soft gene and role from agent conceptually and linguistically; (2) roles exist throughout the whole process of MAS development. To demonstrate our language, we take the case of RoboCup [6] football team simulation. The main development processes in natural language is as follows:

1. Capture use cases and goals;
2. Identify roles from use cases;
3. Construct role organization;
4. For system components, distill their soft genes;
5. For each role, considering soft genes, if the appropriate genes does not exist, then go to 6; else
 - a) Binds roles to genes and generate agents
 - b) Describes dynamic properties of bind relation between agents and roles
 - c) Go to 7
6. Generates agents according to roles; Go to 5.(a).
7. Generates codes for agents with roles bound;

5.1 Capture Use Cases and Goals

Use cases outline the system events and their interactions. We adopt Use Case Diagram in UML to describe use cases. Figure 4 represents a use case (there should be more in real system), in which rectangle denotes Actor and ellipse denotes Use Case. In this example, Goalkeeper actor includes Hand Out use case or Kick Out use case to pass the football; Linebacker actor includes Accept Ball use case to get the football. In some possible conditions, Kick Out use case may be extended by Run With Ball use case. See [17] for detailed information about `<< include >>` and `<< extend >>` stereotypes. Goal hierarchies help to decompose system goal to subgoals. We omit the graph here.

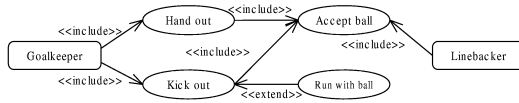


Fig. 4. Example of a use case of football team

5.2 Identify Roles

Roles can be identified from use cases [16,18] and goals. Since they are not sufficient for describing all the roles and events in the MAS. An assistant method is to check the words with -er, -ist or -or suffix in the requirement specification. Figure 5 shows an example notation of role, in which the right top text is the role’s name, the ellipse with text shows the goals the role takes, the middle rectangle with text shows the attributes, the bottom rectangle with text shows the services the role provides and its responsibility. For more information about these concepts, readers refer to [14].

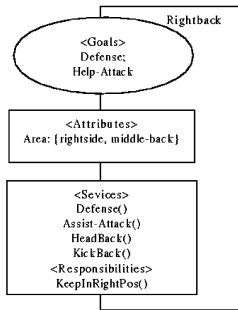


Fig. 5. Example of a role in a football team

5.3 Construct Role Organizations

Roles are not isolate. Every role communicates and interacts with other roles to fulfill its goals. Besides, roles can be specialized or aggregate to other roles. Inheritance and aggregation associations respectively denote the specialization/generalization and aggregation/decomposition relations among roles. Figure 6 shows an example of role organization, in which rectangle with a semi-circle denotes role; arrow line denotes communication path, triangle denotes inheritance relation, diamond denotes aggregation relation, and rectangle with a line on left-top corner denotes organization. For semantics of the notations, readers refer to [19].

Role’s interaction model is sequence diagrams and it may model roles’ interactions well. Just as figure 7 shows.

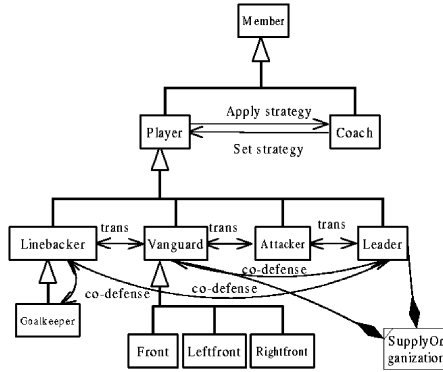


Fig. 6. Example of a role-organization in a football team

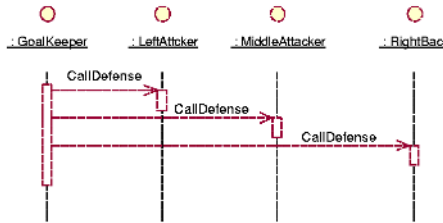


Fig. 7. The example of role interaction in UML

5.4 Distill Soft Genes

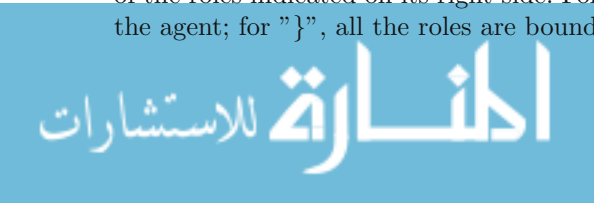
Check system individuals and distill their genes. These soft genes later will be inherited to become other genes. The genes determine agent’s attributes and behavior with rules. Behavior rules are represented as:

$$\langle \text{pre - condition} \rangle \text{ behavior} \langle \text{post - condition} \rangle$$

5.5 Bind Roles to Soft Genes and Generate Agents

For each role, the appropriate soft genes may exist or not. For genes (instances) in existence, they (with the roles) are classified to agent directly. For inexistent soft genes, our insight is that they are related with roles and thus can be generated from roles. For detailed methods, readers are referred to [14].

An agent can change its roles dynamically. To make this property clear, we apply finite automata to describe agent’s role transitions. See the example in Figure 8. Circle represents state, which denotes the roles bound to the agent. Arrow line denotes state transition, which is executed under the conditions or messages listed near the line. The notation "}" / "{" denote OR/AND relation of the roles indicated on its right-side. For "}", only one of the roles is bound to the agent; for "{", all the roles are bound to the agent.



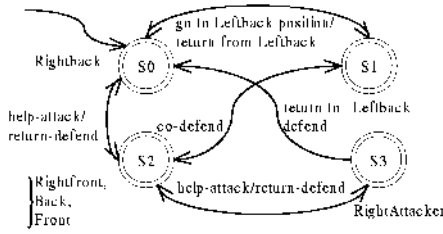


Fig. 8. Example of role transitions of football team

6 Related Works

Role concept has long been concerned since the advent of OO (Object-Oriented), a famous work is [20,4]. Concerning role’s effect in MAS development, Gaia and MaSE methodologies think roles are the result of analysis as undertakers of system goal, and turned into agent classes in the implementation phase, while the concept of role per se disappears. Besides, because of the fixation of role to agent, the interactions among agents are decided according to the system goal in the phase of design. As a result, an agent cannot change its roles at run-time.

The devising of MAS language is also a problem that has puzzled the agent researchers for a long time. The works in existence include AGENT0 [21], SLABS [22], etc. They provide foundations for AO language design. However further researches on the implementation facilities are needed.

7 Conclusions and Future Works

In this paper, we propose a new definition of agent based on soft gene and role. The new definition leads us to a diagrammatic role-based modelling method supporting MAS analysis and design. With RoMAS, the ability and behavior embodied by an agent in MAS is a combination of its soft genes and its roles. Soft genes state the basic perceptive and behavior abilities; roles represent system goal and constrain agents’ behavior. They exist throughout all phases from analysis to implementation, which enables a natural realization of dynamic bindings between agents and roles.

Further work focuses on developing an application with RoMAS:

1. By introducing the concept of soft agent, role and organization to the system, enable every agent in the system to take on roles according to its own mental state and its situation, so as to improve the system’s adaptive and collaborative ability.
2. Based on the study of role concept in MAS and further analysis of its dynamics, take the RoboCup as a case, design and develop a simulation client to testify RoMAS’ pragmatic efficiency.

Acknowledgments. Thanks Mr. James Odell for his helpful advice and discussion. Thanks Ms. LiJun Shan for her correcting English grammar.

References

1. Jennings, N.R.: On agent-based software engineering. *Artificial Intelligence* **117**(2) (2000)
2. Wooldridge, M., Ciancarini, P.: Agent-Oriented Software Engineering: The State of The Art . in *Handbook of Software Engineering and Knowledge Engineering* (2001,World Scientific Publishing)
3. Shoham, Y.: Agent-Oriented Programming. *Artificial Intelligence* **60** (1993) 51–92
4. Wooldridge, M., Jennings, N., Kinny, D.: The Gaia Methodology for Agent-Oriented Analysis and Design. *Journal of Autonomous Agents and Multi-Agent Systems* **3** (2000) 285–312
5. Web: (www.etse.urv.es/recerca/banzai/toni/MAS/)
6. RoboCup: (<http://www.robocup.org>)
7. Yan, Q., Shan, L.J., Mao, X.J.: RoMAS: A Role-Based Modeling Method for Multi-Agent System. *Proceedings of International Conference on Active Media Technology*, World Scientific Publishing (2003)
8. Feng, Z.Y., Sun, C.S., Wang, D.: Actor Theory: New Age and New System Calling New Human Study (in Chinese). Peking University Press (1994)
9. Iglesias, C., Garijo, M., Gonzalez, J.: A Survey of Agent-Oriented Methodologies. *Intelligent Agents V* (1999)
10. DeLoach, S.A., Wood, M.F., Sparkman, C.H.: Multiagent Systems Engineering. *International Journal of Software engineering and Knowledge Engineering* **11**(3) (2001)
11. Staff, A.H.: *The American Heritage Dictionary*. Turtleback Books (01/01/2001)
12. Dawkins, R.: *The SELFISH GENE*. Oxford University Press Paperback (October 1989)
13. Spivey, J.M.: *The Z Notation: A Reference Manual*. Prentice-Hall (1989)
14. Yan, Q., Mao, X.J., Qi, Z.C.: Modeling Role-Based Organization of Agent System. *UKMAS'02* (2002)
15. DeLoach, S., Wood, M.: Developing multiagent systems with agenttool. *Intelligent Agents VI* **1757** (2000)
16. Jacobson, I.: *Object Oriented Software Engineering: A Use Case Driven Approach*. Addison Wesley (1992)
17. OMG: <http://www.omg.org/technology/documents/formal/>. Specification of UML (1995)
18. KenDall, E.A., Palanivelan, U., Kalikivayi, S.: Capturing and Structuring Goals: Analysis Patterns. EuroPlop'98, *European Pattern Languages of Programming* (July 1998)
19. Andersen, E.E.: *Conceptual Modeling of Objects: A Role Modeling Approach*. PhD thesis, PhD Thesis, University of Oslo (1997)
20. Reenskaug, T., Wold, P., Lehne, O.A.: *Working with Objects, The OOram Software Engineering Method* . Manning Publications Co., Greenwich (1996)
21. Shoham, Y.: Agent0: An Agent-Oriented Programming Language and Its Interpreter. *Journal of Object-Oriented Programming* **8**(4) (1991) 19–24
22. Zhu, H.: SLABS: A Formal Specification Language for Agent-Based Systems. *Int. J. of Software Engineering and Knowledge Engineering* **11** (2001) 529–558

Author Index

- Athanasiadis, Ioannis N. 96
Aknine, Samir 138
- Bauer, Bernhard 1
Botti, Vicente 25
Brueckner, Sven 123, 201
- Dyke Parunak, H. Van 123, 201
- Ferber, Jacques 214
Fleischer, Mitch 123
Fuentes, Rubén 110
- Georgousopoulos, Christos 167
Giret, Adriana 25
Gómez-Sanz, Jorge J. 110
Goradia, Hrishikesh J. 153
Gutknecht, Olivier 214
- Hassas, Salima 185
- Jouvin, Denis 185
Juan, Thomas 53
- Karageorgos, Anthony 167
Kehagias, Dionisis 96
Klein, Mark 85
- Manson, G. 69
Mao, XinJun 231
Michel, Fabien 214
- Mitkas, Pericles A. 96
Mouratidis, Haralabos 69
Müller, Jörg P. 1
- Odell, James J. 69, 123, 201
- Pavón, Juan 110
Perini, Anna 36
Pistore, Marco 36
Poggi, Agostino 69
- Qi, ZhiChang 231
Quenum, José Ghislain 138
- Rana, Omer F. 167
Rimassa, Giorgio 69
Roveri, Marco 36
- Sauter, John 201
Slodzian, Aurélien 138
Sterling, Leon 53
Susi, Angelo 36
Symeonidis, Andreas L. 96
- Turci, Paola 69
- Vidal, José M. 153
- Yan, Qi 231
- Zhu, Hong 231