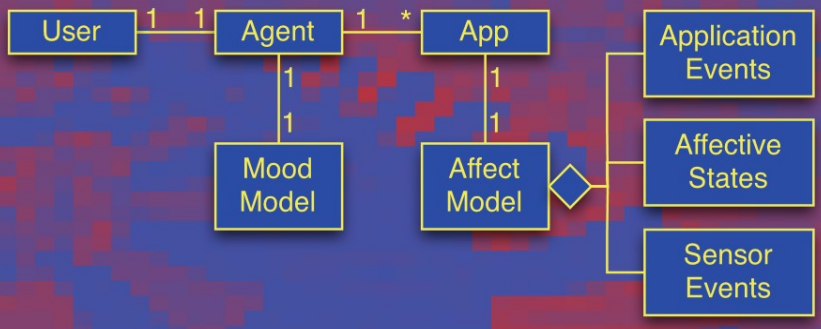
State-of-the-Art Survey

LNCS 6038

Marie-Pierre Gleizes
Jorge J. Gomez-Sanz (Eds.)

Agent-Oriented Software Engineering X

**10th International Workshop, AOSE 2009
Budapest, Hungary, May 2009
Revised Selected Papers**



Springer

Lecture Notes in Computer Science 6038

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

Marie-Pierre Gleizes
Jorge J. Gomez-Sanz (Eds.)

Agent-Oriented Software Engineering X

10th International Workshop, AOSE 2009
Budapest, Hungary, May 11-12, 2009
Revised Selected Papers

Springer

Volume Editors

Marie-Pierre Gleizes
Paul Sabatier University, IRIT, Institute de Recherche
en Informatique de Toulouse
118, Route de Narbonne, 31062, Toulouse, Cedex 9, France
E-mail: marie-pierre.gleizes@irit.fr

Jorge J. Gomez-Sanz
Universidad Complutense de Madrid, Facultad de Informatica
Avda. Complutense s/n, 28040 Madrid, Spain
E-mail: jjgomez@fdi.ucm.es

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-19207-4 e-ISBN 978-3-642-19208-1
DOI 10.1007/978-3-642-19208-1
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2011920739

CR Subject Classification (1998): D.2, I.2.11, F.3, D.1, C.2.4, D.3

LNCS Sublibrary: SL 2 – Programming and Software Engineering

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

Even though a multi-agent system (MAS) developer can count on development environments, frameworks, languages, and methodologies, it is unrealistic to think everything is done. Elementary activities in software engineering, such as testing the developed MAS and ensuring an MAS specification meets the initial requirements, have been barely studied in the agent community, compared with other aspects of development, such as design or implementation. Such gaps need to be identified and filled in properly.

Answers will come via a transfer of knowledge from different areas, not only from software engineering. Mathematics, psychology, sociology, artificial intelligence, to mention a few, can supply the theories and results that characterize an MAS as something different from other pieces of software. Such a mixture of disciplines necessarily needs a glue that puts them together into a body of knowledge, and this glue is agent-oriented software engineering.

As suggested by its name, Agent-Oriented Software Engineering Workshop is a forum for works with a strong engineering component. With this bias, the scope of the workshop is still sufficiently general to host different kinds of contributions, from which we would like to highlight on two: MAS specification languages and development processes.

The AOSE community has invested effort in producing MAS specification languages. A MAS specification, written with one of these languages, uses agent related concepts to describe a problem and its solution. The elements of these specification languages are inspired in agent research, being BDI works the most frequently cited.

While agent specification languages embody the research concepts, the development processes capture the development experience. They identify which are the necessary steps to produce an MAS. This implies generating a MAS specification and indicating how, from this specification, a software product, the actual MAS, can be obtained. A development process suggests ways to organize the development. It contains activities (each one with their inputs, outputs, and software support tools), descriptions of products to obtain, and development roles.

In a sense, both specification languages and development processes have served to move on from laboratory-scale developments to a scope of applicability closer to the standard practices of software engineering. Now, it is possible to address every research result and put it in the context of a development by means of these two elements.

Continuing with this endeavor, the 9th International Workshop on Agent-Oriented Software Engineering (AOSE 2009) took place in Budapest in May 2009 as part of the 8th International Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS 2009). It received 30 submitted papers, which

reflects the continuing interest in the area. All papers were reviewed by at least three reviewers from an international Program Committee of 43 members and 6 auxiliary reviewers, and, as a result, 11 papers were selected to appear in this proceedings volume.

As a novel contribution, this volume includes five additional surveys addressing key areas in development: agent-oriented modelling languages, implementation of MAS, testing of MAS, software processes, and formal methods for the development of MAS. They permit analysis of the current state in the generation of specifications of MAS, the way these specifications can be implemented, how they can be validated, and what steps are necessary to do so.

In structuring this volume, we have organized the papers into four sections. The first deals with multi-agent organizations, which provides a valuable abstraction for agents whose integration in a development method is not trivial. The second section addresses concrete development techniques that enable the transition from specification to code or that validate a concrete implementation. The third section moves one step higher, going beyond the concrete technique and proposing a development method for designing concrete types of systems. The last section introduces the surveys elaborated for this volume. They can be used as an introduction to relevant areas of agent-oriented Software Engineering and as an indicator of the work remaining to be done.

1 Organizations

The first section begins with a paper by Oyenan et al. where a conceptual framework for designing reusable multi-agent organizations is proposed. The framework combines component-oriented and service-oriented principles. Components are autonomous multi-agent organizations that provide or use services as defined by generic interfaces. The composition of these services is possible, leading to reusable components that can be composed to build larger organizations.

The second paper, written by Aldewereld et al., focuses on the structural adaptation to context changes using the OperA organizational model. The paper contributes with a formal representation of organizations and a formal interpretation of the adaptation process. The chosen formalism is an extension of computational tree logic (CTL), called LAO.

The section concludes with a paper by Argente et al. The work proposes a guideline focused on service-oriented open MAS for desiging virtual organizations. The guideline suggests activities along the analysis and design stages of a development based on the organization theory and the service-oriented development approach.

2 Development Techniques

The section starts with a paper by Garcia-Magariño et al. This work introduces a technique for producing model transformations automatically using examples of the expected output of the transformation for some example input models. The

technique is known as model transformation based on examples (MBTE). Its use in the context of an MAS development is illustrated using INGENIAS as a development methodology. As a result, different relevant applications are identified, such as producing preliminary definitions of an agent skill or interactions.

The section continues with a paper on automated testing by Zhang et al. The paper describes an approach to unit testing of plan-based systems, with a focus on automated generation and execution of test cases. The contribution provides an automated process for test case generation and requirements for an environment where the unit tests can be executed. The paper includes a discussion of the implemented system and some results from its use.

The last paper is a contribution from Sudeikat and Renz. The work introduces a systemic modelling level which can complement established modelling approaches. Its purpose is to anticipate the collective system behavior from a collection of designs of autonomous actors. As an application of the approach, the authors introduce the Marsworld case study, where a set of mining activities in a distant planet are addressed.

3 Development Method Proposals

The first work in this section presents Koko-ASM, by Sollenberger and Singh, which is a guideline for building affective applications developed by Sollenberger et al. Koko-ASM identifies what pieces of information are necessary for the instantiation of the Koko architecture, a middle-ware that reduces the burden of incorporating affect into applications. The approach is validated with the development of an application for promoting positive physical behavior in young adults.

Following, Hahn et al. present a mechanism for the generation of agent interaction protocols. The mechanisms are based on a domain-specific modelling language for MAS called DSML4MAS. With DSML4MAS, protocols are specified, and then transformed into executable code. As an example, the authors explain how a contract-net protocol is translated into programming instructions for the Jack agent platform.

The next work is a paper by Nunes et al. focused on multi-agent system product lines. It proposes a domain engineering process for development of this kind of system. Original contributions of this process are the possibility of documenting agent variabilities and tracing agent features. The process is represented using the OMG's software process engineering model and illustrated with the development of a Web application.

The last paper in this section by Bogdanovych et al. is about virtual heritage application construction. It proposes a development process based on the concept of virtual worlds in the Second Life platform and Virtual Institutions technology. As an application example, the paper introduces the City of Uruk project, which intends to recreate the ambience of the city Uruk, situated in Iraq, from the period around 3000 B.C., in the Virtual World of Second Life.

4 State-of-the-Art Survey

The first survey concerns modelling approaches, by Argente et al. The adoption of meta-models by the different agent-oriented methodologies has led to a variety of modelling languages, not always supported by meta-modelling languages. This survey studies these languages addressing issues such as the maturity of the modelling language or its expressive capability.

If modelling is important for specifying the system, the next activity one thinks of is implementing the specification. That is the focus of the second survey, written by Nunes et al. This survey encompasses different implementation approaches, from transformation of specifications to manually coding the specification with conventional methods. The interest of this part is to guide newcomers to the different ways a specification can be realized using agent-oriented approaches.

Lately, there is growing concern on the importance of testing activities in agent-oriented development. Nguyen et. al study this less regarded aspect of development, indicating emergent approaches for testing agent systems and what support a developer can find.

All activities in development necessarily have to be considered within a development process. Method engineering, the topic of the survey by Cossentino et al., has studied these activities and how they can be interleaved to produce a new development process. This survey will be useful for those wanting to know what elements an agent-oriented methodology should regard and how a new methodology can be built using fragments from existing methodologies.

To conclude, the last survey, from El Fallah-Seghrouchni et al., aims to capture the current state of formal method approaches in agent-oriented software engineering research. Some formal methods propose their own development approaches, but others could be reused or integrated within other non-formal approaches. El Fallah-Seghrouchni et al. review briefly a selection of these methods, trying to illustrate how they can help in development and what tool support exists.

September 2010

Marie-Pierre Gleizes
Jorge J. Gomez-Sanz

Organization

Workshop Chairs

Marie-Pierre Gleizes (Co-chair)
Université Paul Sabatier
IRIT, Institut de Recherche en Informatique de Toulouse
118, Route de Narbonne, 31062 Toulouse Cédex, France
Email: Marie-Pierre.Gleizes@irit.fr

Jorge J. Gomez-Sanz (Co-chair)
Facultad de Informática
Universidad Complutense de Madrid
Avda. Complutense, 28040 Madrid, Spain
Email: jjgomez@fdi.ucm.es

Steering Committee

Paolo Ciancarini	University of Bologna
Michael Wooldridge	University of Liverpool
Joerg Mueller	Siemens
Gerhard Weiss	Software Competence Center Hagenberg GmbH

Program Committee

Federico Bergenti (Italy)
Carole Bernon (France)
Olivier Boissier (France)
Juan Antonio Botía (Spain)
Massimo Cossentino (Italy)
Keith Decker (USA)
Scott DeLoach (USA)
Virginia Dignum (The Netherlands)
Cu Duy Nguyen (Italy)
Klaus Fischer (Germany)
Ruben Fuentes (Spain)
Paolo Giorgini (Italy)
Adriana Giret (Spain)
Laszlo Gulyas (Hungary)
Christian Hahn (Germany)
Brian Henderson-Sellers (Australia)
Vincent Hilaire (France)
Tom Holvoet (Belgium)
Vicent Julian Inglada (Spain)
Joao Leite (Portugal)
Juergen Lind (Germany)
Viviana Mascardi (Italy)
Frédéric Migeon (France)
Simon Miles (UK)
Ambra Molesini (Italy)
Haris Mouratidis (UK)
Andrea Omicini (Italy)
H. Van Dyke Parunak (USA)
Juan Pavón (Spain)
Michal Pechoucek (Czech Republic)
Carlos Jose Pereira de Lucena (Brazil)
Anna Perini (Italy)
Gauthier Picard (France)
Alessandro Ricci (Italy)

Fariba Sadri (UK)
Valeria Seidita (Italy)
Onn Shehory (Israel)
Arnon Sturm (Israel)
Viviane Torres da Silva (Brazil)
Laszlo Varga (Hungary)
Danny Weyns (Belgium)
Michael Winikoff (New Zealand)
Eric Yu (Canada)

Auxiliary Reviewers

Estefania Argente
Jiri Hodik
Mirko Morandini
Christophe Sibertin-Blanc
Martin Slota
Jiri Vokrinek

Table of Contents

IV State-of-the-Art Survey

Part I
Organizations

Exploiting Reusable Organizations to Reduce Complexity in Multiagent System Design*

Walamitien H. Oyenan, Scott A. DeLoach, and Gurdip Singh

Multiagent and Cooperative Robotics Laboratory,
Kansas State University 234 Nichols Halls, Manhattan Kansas, USA
{oyenan,sdeloach,gurdip}@k-state.edu

Abstract. Organization-based Multiagent Systems are a promising way to develop complex multiagent systems. However, it is still difficult to create large multiagent organizations from scratch. Multiagent organizations created using current AOSE methodologies tend to produce ad-hoc designs that work well for small applications but are not easily reused. In this paper, we provide a conceptual framework for designing reusable multiagent organizations. It allows us to simplify multiagent organization designs and facilitate their reuse. We formalize the concepts required to design reusable organization-based multiagent services and show how we can compose those services to create larger, more complex multiagent systems. We demonstrate the validity of our approach by designing an application from the cooperative robotics field.

Keywords: Multiagent Organizations, Design Methodologies, Cooperative Robotics.

1 Introduction

Multiagent Systems (MAS) have been seen as a new paradigm to cope with the increasing need for dynamic applications that adapt to unpredictable situations. Large MAS are often composed of several autonomous agents engaging in complex interactions with each other and their environment. Consequently, providing a correct and effective design for such systems is a difficult task. To reduce this complexity, Organization-based Multiagent Systems (OMAS) have been introduced and they are viewed as an effective paradigm for addressing the design challenges of large and complex MAS [9, 25]. In OMAS, the organizational perspective is the main abstraction, which provides a clear separation between agents and system, allowing a reduction in the complexity of the system. To support the design of OMAS, several methodologies have been proposed [8].

Nonetheless, one of the major problems with the wide-scale adoption of OMAS for the development of large-scale applications is that, so far, the methodologies proposed work well for small systems, but are not well suited for developing

* This work was supported by grants from the US National Science Foundation (0347545) and the US Air Force Office of Scientific Research (FA9550-06-1-0058).

M.-P. Gleizes and J.J. Gomez-Sanz (Eds.): AOSE 2009, LNCS 6038, pp. 3–17, 2011.

large and complex applications. Designers generally handle complexity based on intuition and experience, leading to ad-hoc designs that are difficult to maintain.

Therefore, we propose employing reusable multiagent organizations designed using both component-oriented and service-oriented principles. These two well-established software engineering approaches allow us to decompose large multiagent organizations into smaller organizations that can be developed separately and composed later, thus allowing designers to design large and complex OMAS more easily. Herein we assume that organizations are populated by cooperative agents that always attempt to achieve their assigned goals.

In our approach, services are the key concept allowing us to compose OMAS components into larger, more complex systems. *OMAS components* are autonomous multiagent organizations that provide or use services as defined by generic interfaces called *connection points*. OMAS components are composed such that required services match provided services. The composition process ensures the consistency of all the bindings and results into a valid composite organization.

Our contribution is two-fold. First, we rigorously specify the key concepts used in our framework and describe the mechanisms that allow the design of reusable OMAS components. Second, we formally define the composition process through which reusable OMAS components can be composed to build larger organizations.

We present related work in Section 2 followed by an overview of our organizational model in Section 3. Section 4 defines the key concepts of composable organizations while Section 5 describes our composition process. Finally, we illustrate our approach with an example followed by conclusions and future work.

2 Related Work

Several frameworks for multiagent systems have been proposed to deal with the complexity of large software systems [9, 10, 18, 25], while decomposition has long been suggested as a mechanism to deal with complex systems [14]. While current agent-oriented methodologies suggest decomposing organizations, they fail to provide guidance or a rigorous composition process. For instance, Ferber et al. propose partitioning a multiagent system into groups [9] that interact via a gatekeeper agent participating in multiple groups. Unfortunately, they provide no prescription for actually aggregating those groups into a coherent system. Cossentino et al. also propose developing OMAS with a hierarchical structure based on holons [3]. While they divide the system into loosely coupled sub-organizations (holons), there is no guidance on how to recombine them. In general, in order to recombine the organizations, the designer must understand the internal behavior of each sub-organization. Our approach proposes the rigorous specification of interfaces required for composition, thus supporting reusability and maintainability of complex OMAS. Organizations can be developed by different designers and combined later.

Others have proposed compositional approaches for building MAS. DESIRE [1] supports the compositional design of MAS in which components represent agents that can be composed of subcomponents. However, like other component-based frameworks, DESIRE is agent-centric and does not support OMAS.

Few agent-oriented approaches explicitly use service-oriented principles. Most of the unifying work between services and agents concern agent-based service-oriented systems, where agents simply wrap services [12, 13]. However, Cao et. al. propose a methodology called Organization and Service Oriented Analysis and Design (OSOAD) [2], which combines organizational modeling, agent-based design and service-oriented computing to build complex OMAS. Their approach is similar to ours in the sense that complex OMAS are built using service concepts. However, OSOAD services are offered only at the agent level. Our approach permits entire organizations to act as service providers (while still allowing agent-level services), thus allowing us to develop *cooperative services* that cannot be provided by individual agents. In addition, we specify standard interfaces that allow OMAS components to be composed without requiring internal design knowledge.

3 Organizational Model

Our work is based on the Organization Model for Computational Adaptive Systems (OMACS) [7]. OMACS is a formal framework for describing OMAS and is supported by the O-MaSE process framework [10]. OMACS defines an organization as a set of goals (G) that the organization is attempting to accomplish, a set of roles (R) that must be played to achieve those goals, a set of capabilities (C) required to play those roles and a set of agents (A) who are assigned to roles in order to achieve organizational goals. While an agent may play an assigned role in any way it wishes, OMACS does assume the agent will adhere to some minimal expected behavior (e.g. working toward its assigned goal). (There are more entities defined in OMACS that are not relevant for this paper and the reader is referred to [7] for the complete model). In this paper, we use a generalization of the OMACS model and only consider the goals, roles and the relationship that exists between them. Those entities represent the persistent part of the organization, the organization structure [9], which can be populated later with heterogonous agents to produce a concrete organization. There are many other organizational models for OMAS [8] and the approach proposed in this paper could well be adapted to any of them.

In a typical multiagent organization, organizational goals and roles are organized in a goal tree [11, 22, 24] and a role model [15, 24, 25] respectively. We choose to organize our goals using a Goal Model for Dynamic Systems (GMoDS) [6]. In a GMoDS goal model, goals are organized in a goal tree such that each goal is decomposed into a set of subgoals using an OR-decomposition or an AND-decomposition. In addition, the GMoDS goal model contains two time-based relationships between goals: the *precedes* and *triggers* functions.

We say *goal g1 precedes goal g2* if g1 must be satisfied before g2 can be pursued by the organization. Moreover, during the pursuit of specific goals, events may occur that cause the instantiation of new goals. Instantiated goals may be parameterized to capture a context sensitive meaning. If an event e can occur during the pursuit of goal g1 that instantiates goal g2, we say *g1 triggers g2 based on e.* GMoDS defines a goal model *GM* as a tuple $\langle G, Ev, parent, precedes, triggers, root\rangle$ where:

- *G: set of organizational goals (where the set* G_L *represent the leaf goals).*
- *Ev: set of events.*
- *parent:* $G \rightarrow G$ *; defines the parent goal of a given goal.*
- *precedes:* $G \rightarrow 2^G$ *; indicates all the goals preceded by a given goal.*
- *triggers:* $Ev \rightarrow 2^{G \times G}$*;* $\langle g1, g2\rangle \in triggers(e)$ *iff g1 triggers g2 based on e.*
- $root \in G$*; the root of the goal model.*

We organize our roles using a role model that connects the various roles by protocols. There are two types of roles: internal roles and external roles. Internal roles are the typical roles defined inside the organization. External roles are placeholders for roles from an external organization; they represent unknown roles with which the organization must interface. Eventually external roles will be replaced by concrete roles (internal roles) from other organizations. We define our role model *RM* as a tuple $\langle R, P, participants\rangle$ where:

- *R: set of internal and external roles*
- *P: set of protocols*
- *participants:*$P \rightarrow 2^{R \times R}$*; indicates the set of role pairs connected by a protocol*

Finally, we define a multiagent organization *org* as a tuple $\langle GM, RM, achieves, INCP, OUTCP\rangle$ where:

- *GM: Goal Model*
- *RM: Role Model*
- *achieves:* $R \rightarrow 2^{G_L}$ *; indicates the set of leaf goals achieved by given a role.*
- *INCP: the set of entry connection points exposed by the organization (see Section 4.3).*
- *OUTCP: the set of exit connection points exposed by the organization (see Section 4.3)*

4 Service Model

In our framework, services are common functionalities encapsulated in OMAS components. Once designed, OMAS components can be used by other organizations to build larger systems. Fig. 1 shows our metamodel, comprising the service and organizational entities along with their relationships. The central concept is that of *Service.* Services offer one or more *operations.* Each operation possesses a *connector* that is used to connect connection points exposed by

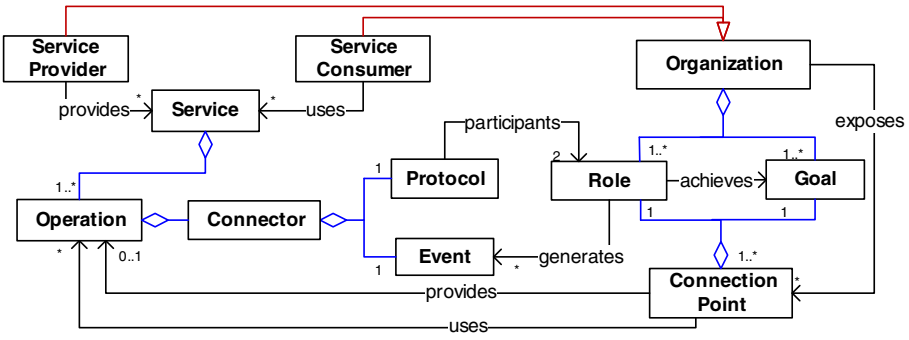


Fig. 1. Organizational Service Metamodel

organizations. *Connection points* are goals and roles that can be bound together by *events* and *protocols* from connectors. *Service providers* and *service consumers* are both autonomous organizations who respectively provide and use operations from services. These entities are discussed in detail in the following subsections.

4.1 Services

A service is a logical entity that represents a coarse-grained multiagent functionality. This coarse-grained functionality is made of a set of fine-grained functionalities, called operations. Each service possesses an *XML-based specification* that contains a description of what the service proposes and provides a specification of each operation provided. To be functional, a service must be implemented by at least one provider. Services facilitate reuse in that they allow consumers to request operations based solely on the service specification.

4.2 Operations and Connectors

An operation represents an implementation of a functionality declared in a service. From an organizational standpoint, we view an *operation* as a set of application-specific organizational goals that an organization needs to achieve in order to reach a desired state. Operations can result in computations (e.g. computing an optimal path for a swarm of UAVs) or actions (e.g. neutralizing an enemy target).

Each operation has a set of *preconditions* and *postconditions*, an *interaction protocol*, and a *request event*. The request event is used to invoke the operation and includes the parameters passed to the operation at initialization. Once the operation is instantiated, the interaction occurs via the interaction protocol, which specifies the legal interactions between consumers and providers. The interaction protocol and the request event form a *connector*, which provides the "glue" that binds consumers and providers together.

4.3 Connection Points

A *connection point* is a logical construct associated with an operation. It is composed of a goal-role pair. There are two types of connection points: entry and exit connection points. An *entry connection point* guarantees a proper execution of the operation it provides. Its goal and role components are called the entry goal and entry role respectively. The set of entry connection points of an organization is denoted by INCP. We say that an entry connection point *provides an operation* if its entry goal is instantiated based on the occurrence of the request event of that operation and its entry role engages with an external role in the interaction protocol defined for that operation. An *exit connection point* guarantees a proper request of the operation it uses. Its goal and role components are called the exit goal and exit role respectively. The set of exit connection points of an organization is denoted by OUTCP. We say that an exit connection point *uses an operation* if its exit goal generates a goal trigger based on the request event of that operation and if its exit role engages with an external role in the interaction protocol defined for that operation (Fig. 2). Hence, a connection point role is an organizational role that would need to be played by an agent in order to achieve the corresponding connection point goal.

4.4 Service Providers

A *service provider* is an organization (OMAS component) that provides all the operations of a particular service. We say that an organization provides a service if, for all operations of the service, it exposes a unique entry connection point providing that operation. In addition, a service provider needs to be designed such that whenever an operation is requested and its preconditions are met, it pursues a set of its goals whose achievement satisfies the postconditions of that operation. As for any goal failures in OMACS, operation failures result in an autonomous reorganization in an attempt to overcome the failure [17].

4.5 Service Consumer

A *service consumer* is an organization (OMAS component) that uses one or more operations from various services. To use an operation, an organization needs to exposes at least one exit connection point using that operation. For each operation used, service consumers can expose multiple connection points. Designers of service consumers choose the operations they need based on the service specification that describes what each of its operation is suppose to do.

5 Composition of Services

Composition is a design-time process that binds a consumer organization with a provider in order to create a single composite organization. This process is illustrated in Fig. 2. Basically, given an operation, the composition process

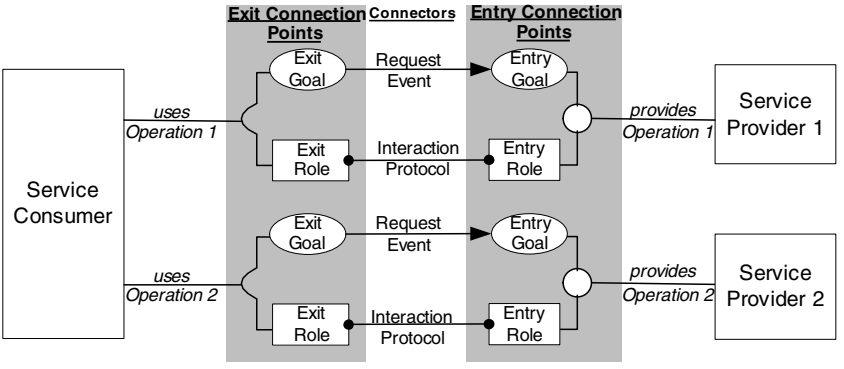


Fig. 2. Composition of Organizations through connection points and connectors

connects the exit connection point of a consumer to the entry connection point of a provider using the operation's connector. This interconnection ensures that the consumer organization can invoke the operation via the request event and that both organizations can interact via the interaction protocol. Formally, the *composition of organizations* org_1 *with* org_2 *over a connection point* cp_1 *requiring an operation op* is defined whenever cp_1 is an exit connection point from org_1 using op and org_2 exposes a connection point cp_2 providing op. This composition is denoted $org_1 \vdash^{cp_1,op} org_2$.

Sometimes, designers may want to compose all exit connection points using the same operation from only one provider. Thus, we define the composition of two organizations over an operation as their successive compositions over all the exit connection points requiring that operation. Hence, for all connection points cp_i from org_1 using an operation op, we have:

$$org_1 \vdash^{op} org_2 = (...((org_1 \vdash^{cp_1,op} org_2) \vdash^{cp_2,op} org_2) \vdash^{\cdots} ... \vdash^{cp_n,op} org_2).$$

The composition process is iterative and continues until the resulting composite organization requires no more operations. The result is a standalone application that uses no external services. Having a single organization simplifies reorganization tasks by allowing us to reuse existing work concerning reorganization of single organizations [20, 21, 26].

Next, we formally define the composition process through which reusable OMAS components can be composed to build larger organizations. We have a proof sketch that shows this composition will always be correct under certain conditions, but space would not permit us to put any details in the paper.

Given two organizations $org_1 = \langle GM_1, RM_1, achieves_1, INCP_1, OUTCP_1 \rangle$, $org_2 = \langle GM_2, RM_2, achieves_2, INCP_2, OUTCP_2 \rangle$, an operation *op* and two connection points cp_1 from org_1 and cp_2 from org_2 such that cp_1 *uses op* and cp_2 *provides op*. Given that $org_3 = \langle GM, RM, achieves, INCP, OUTCP \rangle$, such that $org_3 = org_1 \vdash^{cp_1,op} org_2$, we define the composite organization org_3 in the next subsections.

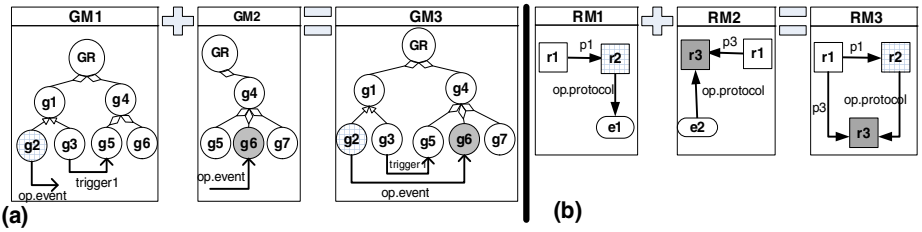


Fig. 3. Merging goal models (a) and role models (b)

5.1 Goal Model Composition

Without loss of generality, we assume that all goal models have the same root, which is an AND-decomposed goal called the generic root (GR). Moreover, we consider that two goals are equal if they are identical and their parents are identical. This definition of equality of goals ensures that the union of two goal trees is a tree instead of a graph. Given two goal models $GM_1 = \langle G_1, Ev_1, parent_1, precedes_1, triggers_1, GR\rangle$, and $GM_2 = \langle G_2, Ev_2, parent_2, precedes_2, triggers_2, GR\rangle$, we define the composite goal model $GM = \langle G, Ev, parent, precedes, triggers, root\rangle$ such that:

root $= GR$, $\quad$ ***G*** $= G_1 \cup G_2$, $\quad$ ***Ev*** $= Ev_1 \cup Ev_2$,
parent: $\forall g \in G, parent(g) = parent_1(g) \cup parent_2(g)$,
precedes: $\forall g \in G, precedes(g) = precedes_1(g) \cup precedes_2(g)$,
triggers: $\forall e \in Ev$, *triggers(e)*=

$$= \begin{cases} triggers_1(e) \cup triggers_2(e) & \text{if } e \neq op.event, \\ triggers_1(e) \cup triggers_2(e) \cup \{(cp_1.goal, cp_2.goal)\} & \\ \quad -\{(cp_1.goal, \varnothing), (\varnothing, cp_2.goal)\} & \text{if } e = op.event. \end{cases}$$

Note that cp_1.goal is an exit goal in GM_1 and cp_2.goal is an entry goal in GM_2. The composition is illustrated in Fig. 3a, where g2 is an exit goal and g6 is an entry goal.

5.2 Role Model Composition

Given $RM_1 = \langle R_1, P_1, participants_1\rangle$, $RM_2 = \langle R_2, P_2, participants_2\rangle$, let e_1 and e_2 be two external roles such that $(cp_1.role, e_1) \in participants_1(op.protocol)$ and $(e_2, cp_2.role) \in participants_2(op.protocol)$, where cp_1.role is an exit role in RM_1 and cp_2.role is an entry role in RM_2. We define $RM = \langle R, P, participants\rangle$ such that:

R $= R_1 \cup R_2 - \{e_1, e_2\}$, $\quad$ ***P*** $= P_1 \cup P_2$,
participants: $\forall p \in P$, *participants(p)*=

$$= \begin{cases} participants_1(p) \cup participants_2(p) & \text{if } p \neq op.protocol, \\ participants_1(p) \cup participants_2(p) \cup \{(cp_1.role, cp_2.role)\} & \\ \quad -\{(cp_1.role, e_1), (e_2, cp_2.role)\} & \text{if } p = op.protocol. \end{cases}$$

The composition of role models we have just described is illustrated in Fig. 3b. In this figure, role r2 is an exit role and role r3 is an entry role.

5.3 Organization Composition

Finally, to complete org_3, we need to define the *achieves* function along with the connection points. The *achieves* function is defined as:

$$\textbf{achieves(r)} = achieves_1(r) \cup achieves_2(r), \quad \forall r \in R.$$

The sets of entry and exit connection points exposed by org_3 are:

$$\boldsymbol{INCP} = INCP_1 \cup INCP_2 - \{cp_2\}.$$
$$\boldsymbol{OUTCP} = OUTCP_1 \cup OUTCP_2 - \{cp_1\}.$$

6 Case Study

To demonstrate the validity of our framework for designing OMAS, we design an application called Cooperative Robotic for Airport Management (CRAM). In this application, a team of heterogeneous robots is in charge of handling some aspects of the airport management task. Essentially, the team needs to clean the building and perform cargos inspections. Suspicious cargos are sent to another location for further inspection.

In our framework, OMAS components can be present in a repository or come from the decomposition of the current problem. For this example, we develop one service, the *cleaning service*, and explain how it can be used to develop our CRAM application.

In the organization models presented in this example (Fig. 4, Fig. 5, and Fig. 6), *goals* are shown as ovals, *internal roles* as rectangles, *external roles* as round rectangles, *precedes* and *triggers* functions as open-head arrows, *protocols* as full-head arrows and *achieves* functions as dashed lines. *Conjunctive goals* are connected to their subgoals by diamond-shaped links and *disjunctive goals*

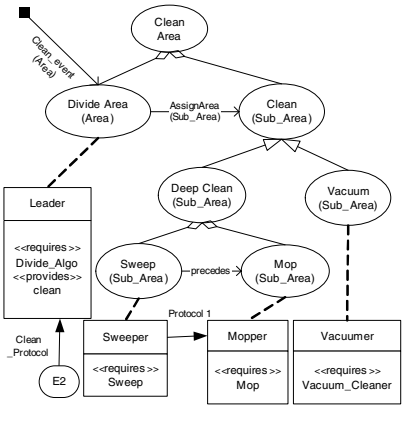


Fig. 4. Cooperative Cleaning organization model

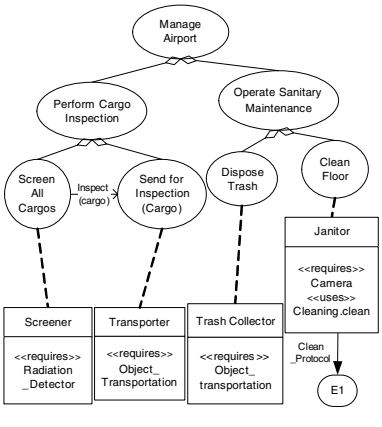


Fig. 5. CRAM organization model

by triangle-shaped links. *Entry goals* are identified by being the destination of a trigger that has no source. *Exit goals* are always the leaf goals achieved by exit roles. In the role models, agent capabilities [7] are identified by the keyword *'requires'*. Entry roles specify operations provided by using the keyword *'provides'* while exit roles specify operations required by the keyword *'uses'*. Due to space limits, we do not discuss aspects of the organization irrelevant to our approach.

6.1 The Cleaning Service

Cooperative cleaning is a common problem in cooperative robotics and several works have been published regarding the use of robots for cleaning [16, 19, 23]. Here, we propose a *Cleaning Service* whose main operation is to clean a given area. We design the *Cooperative Cleaning Organization*, shown in Fig. 4, which involves a team of robots coordinating their actions to clean an area. Hence, this OMAS component provides the *Cleaning Service*. The entry connection point providing the *clean* operation is made of the goal *Divide Area* and the role *Leader*. The *Divide Area* goal is in charge of dividing an area into smaller areas that can be handled by individual robots. Once the area to be cleaned has been divided, the *Clean* goal is triggered. The *Clean* goal is decomposed into two disjunctive goals. Hence, it offers two ways of cleaning; the organization can decide to either do a deep clean (*Deep Clean* goal) or just vacuum (*Vacuum* goal). The *Deep Clean* goal is further decomposed into two conjunctive goals: *Sweep* and *Mop*.

6.2 The Cooperative Robotic for Airport Management Organization

Next, we build the CRAM organization that uses the Cleaning Service. Its design is presented in Fig. 5. The main goal of the system, *Manage Airport*, has two conjunctive subgoals that represent the two main tasks of our system: *Perform Cargo Inspection*, *Operate Sanitary Maintenance*. Those goals are in turn further decomposed into conjunctive leaf goals. For each leaf goal in the CRAM organization, we design a role that can achieve it. Moreover, we identify that the *Janitor* role can use the *Cleaning Service* for the achievement of the *Clean Floor* goal. Thereby, the organization created contains the exit connection point (identified as goal-role pair): $\langle CleanFloor, Janitor \rangle$.

6.3 The Composition Process

In this section, we compose the CRAM application with the cleaning component in order to obtain a single composite organization. The CRAM uses the *clean* operation from the *Cleaning Service* that is provided by the *Cooperative Cleaning* organization. Let *cram* and *cleaning* be the CRAM and Cooperative Cleaning organizations respectively and let *cp_Janit* and *cp_Lead* be the connection

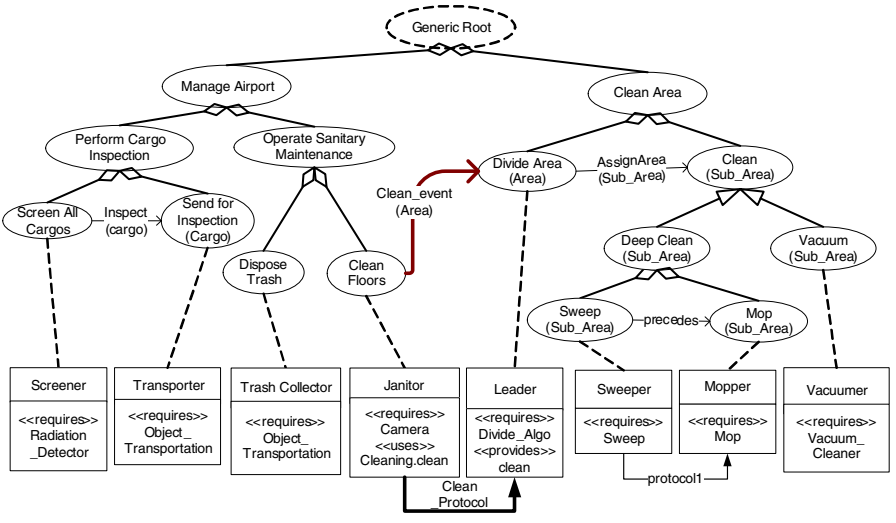


Fig. 6. Organization model obtained by composition of cram and cleaning over the clean operation

points $\langle CleanFloor, Janitor\rangle$ from *cram* and $\langle DivideArea, Leader\rangle$ from *cleaning* respectively. We have:

$$\textbf{\textit{cleaning}} = \langle gm\_svc, rm\_svc, achieves\_svc, incp\_svc, outcp\_svc\rangle \ ,$$

where goal model *gm_svc*, role model *rm_svc* and achieves function *achieves_svc* are defined as described in Fig. 4, entry connection points set $incp\_svc = \{cp\_Lead\}$, and exit connection points set $outcp\_svc = \{\}$.

$$\textbf{\textit{cram}} = \langle gm\_app, rm\_app, achieves\_app, incp\_app, outcp\_app\rangle,$$

where goal model *gm_app*, role model *rm_app* and achieves function *achieves_app* are defined as described in Fig. 5, entry connection points set $incp\_app = \{\}$, and exit connection points set $outcp\_app = \{cp\_Janit\}$.

By composing *cram* with *cleaning* over operation *clean*, we have:

$$cram \vdash^{clean} cleaning = cram \vdash^{cp\_Janit,clean} cleaning = cram\_clean,$$

such that $\textbf{\textit{cram\_clean}} = \langle gm, rm, achieves, incp, outcp\rangle$, where:

gm, rm, achieves are defined as described in Fig. 6,

$incp = incp\_app \cup incp\_svc - \{cp\_Lead\} = \{\}$,

$outcp = outcp\_app \cup outcp\_svc - \{cp\_Janit\} = \{\}$.

Hence, by composing the *cram* and *cleaning* organizations (Fig. 4 and Fig. 5) over the *clean* operation specified in the *Cleaning Service*, we obtain the composed organization *cram_clean* modeled in Fig. 6.

6.4 Implementation Overview

In this section, we give a brief overview of how the composite organization is implemented. Our implementation is based on design models obtained

from the Organization-based Multiagent System Engineering (O-MaSE) process framework [10], which allows designers to create custom agent-oriented development processes. The implementation of the CRAM organization is based on several design models: the goal model and role model obtained from the composition process as shown in Fig. 6, and an agent model, a capability model, a protocol model and a plan model. However, depending on the actual process used, some of those models are optional [10]. The Capability Model identifies the capabilities and specifies their actions on the environment as state machines. The Agent Model captures agent types, along with the capabilities they posses. The Protocol Model specifies all the protocols that will be executed by agents enacting their assigned roles. Finally, the Plan Model provides a plan that agents execute in order to play a certain role. Plans are expressed as state machines.

At runtime, four main components allow the system to be autonomous and adapt to its environment. Those components are part of the *Control Component* of an agent (Fig. 7). The *Goal Reasoning* takes the goal model as input and tells the system which goals are active and can be pursued by the organization. Details about this component can be found in [6]. The *Organization Model* stores all the knowledge about the structure of the organization [5]. The *Reorganization Algorithm* takes as input a set of active goals and returns a set of assignments [26]. An assignment assigns an agent to play a role in order to achieve a particular goal. Assignments are made based on the capability possessed by agents and some utilities functions. Once an agent has been assigned to play a role, it follows the role's plan in order to achieve its goal. Further details about how assignments are made can be found in [7]. The Goal Reasoning component and the Reorganization algorithm can be implemented in a centralized or distributed

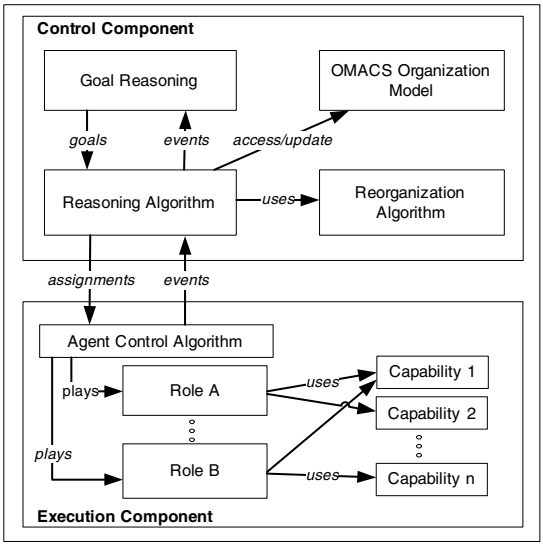


Fig. 7. Organization-based Agent Architecture

way. In a centralized version, only one agent is in charge of the organization. In a distributed version, all agents participate in the organization decision process. Finally, the *Reasoning Algorithm* interacts with the other components in order to make decisions about the next state of the organization [5]. All decisions taken at the *Control Component* level are passed on to the *Execution Component* that is in charge of executing the appropriate roles while using the required capabilities (Fig. 7).

For brevity, we only describe partially the runtime of the model shown in Fig. 6. Detailed implementations of similar OMACS-based systems have been presented in our previous work [7, 17]. During the execution, the goal *Clean Floor* eventually becomes active. A reorganization then occurs which results in the assignment of an agent (say the *Maintenance* agent) to play the *Janitor* role to achieve the newly created *Clean Floor* goal. During the enactment of the *Janitor* role, the event *clean_event* (cf. Fig. 6) occurs whenever an area that needs to be cleaned is detected. This event results in the creation of the goal *Divide Area*, which is the entry point of the clean service. Hence, the trigger acts as a request for the clean service. After reorganization, the *Divide Area* goal is assigned to an agent (say *Clean Manager* agent) playing the *Leader* role. The *Clean Manager* agent plays its role by dividing the area to be cleaned into subareas. Each subarea will be eventually assigned to an agent and when all agents complete, the area is clean. The *Manager* agent (part of the Service Provider organization) can then notify the *Maintenance* agent (part of the Service Consumer organization) through the protocol *clean_protocol* (cf. Fig. 6). At the same time, the *Maintenance* agent can continue looking for other areas to clean and request the cleaning service as often as necessary.

7 Conclusion and Future Work

We have presented an approach to support the development of complex OMAS by developing reusable OMAS components. While many current approaches use decomposition to support separation of concerns at design, they lack effective support for the composition process to recombine the sub-organizations. Our approach defines a composition framework that allows MAS designers to reuse predefined OMAS modeled as components. We described how to design OMAS components so they expose the necessary interfaces to potential consumers so they can request the desired operations. Moreover, we presented a composition process that combines multiple multiagent organizations into a single organization. Finally, we illustrated our approach by composing a Cooperative Robotic Airport Management application with a multiagent cleaning service to produce a single complete system design.

A significant advantage of our approach is the ability to compose multiagent organizations to develop a wide variety of complex applications. In addition, independent OMAS components are easily modifiable, offer a better approach to reusability of design models, help reduce development time and provide better structuring of large and complex MAS. Designers have more flexibility as service providers can easily be replaced with little or no change in the core organization.

We are currently working on extending the O-MaSE process framework [10] and the agentTool (aT^3) development environment [4] to support the systematic design of OMAS by using our compositional approach. We are also interested in offline exploration of alternative designs created using different service providers and verifying them for robustness, flexibility and efficiency. Having predictive data on the quality of alternate designs will help designers choose the best service providers for their applications.

References

1. Brazier, F.M.T., et al.: DESIRE: Modelling Multi-Agent Systems in a Compositional Formal Framework. IJCIS 6(1), 67–94 (1997)
2. Cao, L., Zhang, C., Zhou, M.: Engineering Open Complex Agent Systems: A Case Study. IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews 38(4), 483–496 (2008)
3. Cossentino, M., et al.: A holonic metamodel for agent-oriented analysis and design. In: Mařík, V., Vyatkin, V., Colombo, A.W. (eds.) HoloMAS 2007. LNCS (LNAI), vol. 4659, pp. 237–246. Springer, Heidelberg (2007)
4. DeLoach, S.A.: The agentTool III Project, http://agenttool.cis.ksu.edu/ (cited 2009)
5. DeLoach, S.A.: OMACS: a Framework for Adaptive, Complex Systems. In: Dignum, V. (ed.) Multi-Agent Systems: Semantics and Dynamics of Organizational Models. IGI Global, Hershey (2009)
6. DeLoach, S.A., Miller, M.: A Goal Model for Adaptive Complex Systems. In: International Conference on Knowledge-Intensive Multi-Agent Systems (KIMAS 2009), St. Louis, MO, October 11-14 (2009)
7. DeLoach, S.A., Oyenan, W.H., Matson, E.: A capabilities-based model for adaptive organizations. Autonomous Agents and Multi-Agent Systems 16(1), 13–56 (2008)
8. Estefania, A., Vicente, J., Vicente, B.: Multi-Agent System Development Based on Organizations. Electronic Notes in Theoretical Computer Science 150(3), 55–71 (2006)
9. Ferber, J., Gutknecht, O., Michel, F.: From agents to organizations: An organizational view of multi-agent systems. In: Giorgini, P., Müller, J.P., Odell, J.J. (eds.) AOSE 2003. LNCS, vol. 2935, pp. 443–459. Springer, Heidelberg (2004)
10. Garcia-Ojeda, J.C., et al.: O-maSE: A customizable approach to developing multiagent development processes. In: Luck, M., Padgham, L. (eds.) Agent-Oriented Software Engineering VIII. LNCS, vol. 4951, pp. 1–15. Springer, Heidelberg (2008)
11. Huget, M.P.: Representing Goals in Multiagent Systems. In: Proc. 4th Int'l Symp. Agent Theory to Agent Implementation, pp. 588–593 (2004)
12. Huhns, M.N., Singh, M.P.: Service-oriented computing: key concepts and principles. IEEE Internet Computing 9(1), 75–81 (2005)
13. Huhns, M.N., et al.: Research Directions for Service-Oriented Multiagent Systems. IEEE Internet Computing 9(6), 65–70 (2005)
14. Jennings, N.R.: An agent-based approach for building complex software systems. Commun. ACM 44(4), 35–41 (2001)
15. Juan, T., Pearce, A., Sterling, L.: ROADMAP: extending the gaia methodology for complex open systems. In: Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems: Part 1, pp. 3–10. ACM, Bologna (2002)

16. Luo, C., Yang, S.X.: A real-time cooperative sweeping strategy for multiple cleaning robots. In: Proceedings of the 2002 IEEE International Symposium on Intelligent Control, pp. 660–665 (2002)
17. Oyenan, W.H., DeLoach, S.A.: Design and Evaluation of a Multiagent Autonomic Information System. In: Proceedings of the 2007 IEEE/WIC/ACM International Conference on Intelligent Agent Technology (2007)
18. Padgham, L., Winikoff, M.: Prometheus: A methodology for developing intelligent agents. In: Giunchiglia, F., Odell, J.J., Weiss, G. (eds.) AOSE 2002. LNCS, vol. 2585, pp. 174–185. Springer, Heidelberg (2003)
19. Parker, L.E.: ALLIANCE: an architecture for fault tolerant multirobot cooperation. IEEE Transactions on Robotics and Automation 14(2), 220–240 (1998)
20. Harmon, S.J., et al.: Leveraging Organizational Guidance Policies with Learning to Self-Tune Multiagent Systems. In: The Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems (2008)
21. Sims, M., Corkill, D., Lesser, V.: Automated organization design for multi-agent systems. Autonomous Agents and Multi-Agent Systems 16(2), 151–185 (2008)
22. van Lamsweerde, A., et al.: The KAOS Project: Knowledge Acquisition in Automated Specification of Software. In: Proceedings AAAI Spring Symposium Series, pp. 59–62 (1991)
23. Wagner, I.A., et al.: Cooperative Cleaners: A Study in Ant Robotics. Int. J. Rob. Res. 27(1), 127–151 (2008)
24. Wood, M.F., DeLoach, S.A.: An overview of the multiagent systems engineering methodology. In: Ciancarini, P., Wooldridge, M.J. (eds.) AOSE 2000. LNCS, vol. 1957, pp. 207–221. Springer, Heidelberg (2001)
25. Zambonelli, F., Jennings, N.R., Wooldridge, M.: Developing multiagent systems: The Gaia methodology. ACM Trans. Softw. Eng. Methodol. 12(3), 317–370 (2003)
26. Zhong, C., DeLoach, S.A.: An Investigation of Reorganization Algorithms. In: International Conference on Artificial Intelligence (IC-AI 2006). CSREA Press, Las Vegas (2006)

A Formal Specification for Organizational Adaptation

Huib Aldewereld, Frank Dignum, Virginia Dignum, and Loris Penserini

Institute of Information and Computing Sciences
Universiteit Utrecht - P.O. Box 80089, 3508 TB Utrecht - The Netherlands
{huib,dignum,virginia,loris}@cs.uu.nl

Abstract. Agent organizations are a good means to guarantee certain system objectives in the context of autonomous, self adapting agents. However, in highly dynamic environments such as in crisis management where different organizations have to cooperate flexibly and efficiently, the organizational structure itself should also be adaptable to the circumstances. In this paper we distinguish different types of context changes and the way organizations can adapt to such changes. We give a formal description —of both organizations and the changes— which facilitates proving that certain features remain stable throughout the life cycle of the organization or are justifiable changes.

1 Introduction

Software system development for complex applications requires approaches that take into account organizational and social issues, such as control, dependencies and structural changes. Often, the agent paradigm is proposed for these situations [2,18,11,15]. Nevertheless, typical agent-oriented software engineering approaches are often not well suited to deal with organizational changes and the resulting need for adaptation. Adaptation issues are often ignored, only treated implicitly, or require complete (offline) system redesign. In particular, only a few methodologies have considered organizational and social abstractions as key architectural and coordination requirements for multi-agent systems (MAS), e.g., see [2,18,15].

Following the idea of autonomic computing publicized by IBM [8,12], recent software engineering approaches have focused on the design of single (agent-based) systems with the ability to cope with context changes. However, in complex domains, adaptation can be easier and better studied and handled at an organizational level of abstraction [10]. In this paper, we propose ways to deal with the challenge of adaptation at the organizational level within a software development framework.

The work reported in this paper is part of the general development framework within the European FP7 project ALIVE. As illustration, we adopt a simplified version from the Crisis Management case studied in ALIVE. Due to the high variability of social dependencies and responsibilities among roles, these scenarios are very suitable to study adaptation to context changes.

We use the OperA framework [2,14] that provides social abstractions to study and design organizations of agent societies. The OperA model includes three components:

M.-P. Gleizes and J.J. Gomez-Sanz (Eds.): AOSE 2009, LNCS 6038, pp. 18–31, 2011.

the *organizational model*, the *social model*, and the *interaction model*. This paper focuses on structural adaptation to context changes, using the OperA organizational model as basis for analysis and design. We provide a formal interpretation of the adaptation process for organizations, based on the *Logic for Agent Organizations* (LAO) [3,5].

The aim of our work is modeling context changes that affect organizational structures and agents' social and intentional relationships. The formal specification of organizations enables the description and analysis of adaptation which contributes to the (automatic) redesign of structures.

The paper is structured as follows. Section 2 describes the basic notions of the formal specification for agent organizations adopted in our design framework, and gives an overview about the OperA methodology, applying it to a crisis management scenario. Section 3 discusses different forms of adaptation that may affect organization structures. Section 4 defines the adaptation of organization structures by the use of a formal specification. In Section 5 some related work is given. Finally, Section 6 gives some conclusions and points out main future work directions.

2 Background

2.1 The ALIVE Project

Currently, there is a general feeling that networked applications based on the notion of software services will make it possible to radically create new types of software systems. However, the deployment of such applications will require profound changes in the way in which software systems are designed, deployed and managed.

The ALIVE project [1] aims to contribute to this development by the application of strategies used today to organize the vastly complex interdependencies found in human social, economic behaviour to the structuring and design of future service-based software systems. More specifically, the project aims to combine coordination and organisational mechanisms and Model Driven Design to create a framework for software and service engineering for "live" open systems of active services.

The project extends current trends in service-oriented engineering by adding three extra levels (see Figure 1).

- The *Service level* augments and extends existing service models with semantic descriptions to make components aware of their social context and of engagement rules with other services.
- The *Coordination level* provides the means to specify, at a high level, interaction patterns between services, using a variety of powerful coordination techniques from recent European research in the area.
- The *Organisation level* provides context for the other levels – specifying the organizational rules that govern interactions and using recent developments in organizational dynamics to allow the structural adaptation of distributed systems over time.

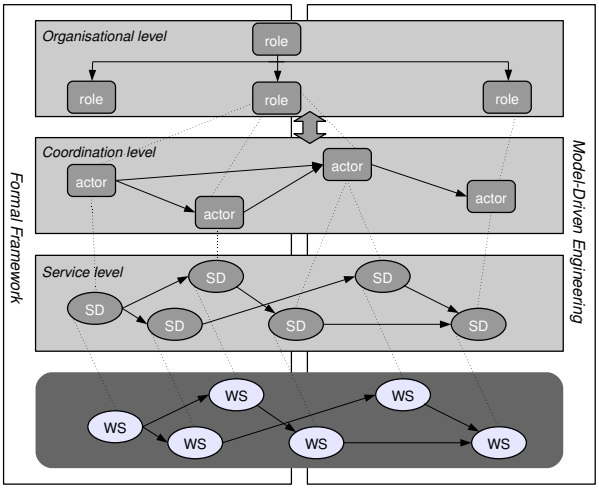


Fig. 1. The ALIVEframework for software and service engineering

2.2 A Formal Model for Organizations

As basis for the specification of organizational adaptation, we use the formal modal Logic for Agent Organizations – LAO [3]. LAO provides a formal definition of organizations, based on the main concepts of structure, environment and agent [17]. In LAO, agents are seen as actors that make possible the achievement of organizational objectives[1]. LAO is an extension of the well known branching time temporal logic CTL [7]. It includes the CTL modalities $\Box$ ('always in the future'), $\Diamond$ ('eventually in the future'), $\circ$ ('in the next state') extended with modalities for agent *ability*, *capability*, *attempt* and *activity* which we discuss below.

Given a world $w \in W$ that represents a possible state of affairs, an organization O^w is defined as

$$O^w = \{A_O^w, \leq_O^w, D_O^w, S_O^w\}$$

where A_O is a set of agents, $\leq_O$ a partial order relation on A_O reflecting the social dependencies among actors of the organization, D_O is a set of objectives (states of affairs to achieve), and S_O is the set of current assets and characteristics of O, e.g., capabilities, activities, responsibilities, etc.

Given a set of atomic propositions Φ, for each agent a we partition Φ in two classes: the set of atomic propositions that agent a is capable of realizing, C_a ($\subseteq \Phi$), and the set of atomic propositions that a cannot realize, $\bar{C}_a$, $\bar{C}_a = \Phi \setminus C_a$. The composition of atomic propositions of an agent $a \in A_O$, i.e., the set of its complex propositions, is called Σ_a.

[1] Even though organizational structures are typical defined in terms of roles, specifying and distributing responsibilities and objectives among roles, for simplicity sake [3] abstracts from the role concept, only considering role enacting agent, an agent that plays one role.

Definition 1. ***(Agent Capability)***
Given formula φ in the language and an agent $a \in A_O$, agent a is capable of φ, represented by $C_a\varphi$ iff $\not\models \varphi$ and $\exists \psi \in \Sigma_a$, such that $\models \psi \rightarrow \varphi$.

Intuitively, $C_a\varphi$ means that a can control φ, which means that a is able (under certain conditions) to make it the case that φ. Note that, the notion of capability gives only a static description of an agent's ability, namely, the necessary condition to realize φ. At any given moment, only the combination of the capability plus the opportunity (e.g., preconditions) leads to the agent's ability, represented by $G_a\varphi$. The opportunity is represented in the semantics by a set of all transitions in which a takes part (influences the result) T_{aw}.

Moreover, while the agent ability ($G_a\varphi$) allows the agent to achieve φ in a next state, it does not express that the agent will act to achieve φ. The latter is named *agent attempt*, represented by $H_a\varphi$. Finally, we introduce the definition of agent activity, *stit* (see to it that). Formally,

Definition 2 (Agent Ability, Attempt and Activity). *Given an agent $a \in A$, agent ability, $G_a\varphi$, agent attempt, $H_a\varphi$, agent control, $IC_a\varphi$, and agent activity, $E_a\varphi$, are defined as:*

G_a: $w \models G_a\varphi$ *iff* $w \models C_a\varphi$ *and* $\exists(w, w') \in T_{aw} : w' \models \varphi$
H_a: $w \models H_a\varphi$ *iff* $w \models G_a\varphi$ *and* $\forall(w, w') \in T_{aw} : w' \models \varphi$
IC_a: $w \models IC_a$ *iff* $\forall w' : (w, w') \in R \Rightarrow (w, w') \in T_{aw}$
E_a: $w \models E_a\varphi$ *iff* $w \models H_a\varphi \wedge IC_a$

Agent *activity* is represented by $E_a\varphi$. Intuitively, $E_a\varphi$ represents the actual action of bringing φ about, namely, agent a can 'cause' φ to be true in all the next states from the current one. The formalization of $E_a\varphi$ is based on the *stit* operator, originally defined in [16].

Another important aspect to be considered is the way organizations deal with complex objectives (which are often not achievable by any single agent) that require the coordination of different agents according to their skills and responsibilities. In the definition of organization above, the dependency between two agents (a and b), represented by $a \leq_O b$, indicates that a is able to delegate some state of affairs to agent b. The dependency relation also allows an agent for delegation of responsibility, as follows.

Definition 3. ***(Agent Responsibility)***
Given an organization $O = \{A_O, \leq_O, D_O, S_O\}$ and an agent $a \in A_O$, the responsibility $R_a\varphi$ is defined as: $R_a\varphi \equiv \Diamond H_a\varphi \vee \Diamond H_a R_b\varphi$, for some $b \in A_O$.

Informally, $R_a\varphi$ means that a has to make sure that a certain state of affair is (eventually) achieved, either by realizing it itself or by delegating that result to others. Notice that, responsibility does not guarantee the achievement of the underlying state of affair. From the above considerations, the *structural dependency* ($C_a R_b\varphi$) is straightforward: $if(a \leq_O b)$ then $C_a R_b\varphi$, namely, a is capable to delegate φ to b.

Finally, we consider the formal specification of structural changes. Re-organization activities imply changes within the organizational structure, such as modifications over objectives, agents, and structural dependencies. Within the formal model, these modifications are represented as follows [5].

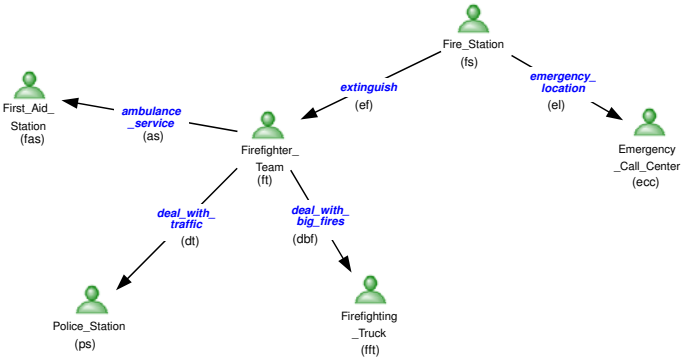


Fig. 2. Social Structure diagram: Fire Station organization (O) example

Definition 4. *In the following reorganization operations in* $O = \{A_O, \leq_O, D_O, S_O\}$ *are presented:*

- ***Staffing:*** *Changes on the set of agents: adding new agents, or deleting agents from the set. Corresponding to personnel activities in human organizations (hiring, firing and training). Represented by* $staff^+(O, a)$ *and* $staff^-(O, a)$.
- ***Structuring:*** *Changes on the ordering structure of the organization. Corresponding to infrastructural changes in human organizations: e.g. changes in composition of departments or positions. Represented by* $struct^+(O, a \leq b)$ *and* $struct^-(O, a \leq b)$.
- ***Strategy:*** *Changes on the objectives of the organization: adding or deleting desired states. Corresponding to strategic (or second-order) changes in human organizations: modifications on the mission, vision, or charter of the organization. Represented by* $strateg^+(O, d)$ *and* $strateg^-(O, d)$.
- ***Duty:*** *Changes the responsibilities in the organization in correspondence to duty assignments in human relations. Represented by* $duty^+(O_i, a, \varphi)$ *and* $duty^-(O_i, a, \varphi)$
- ***Learn:*** *Changes the knowledge (state of affairs) of the organization in correspondence to the change of experiences, knowledge and learning in human organizations Represented by* $learn^+(O_i, \varphi)$ *and* $learn^-(O_i, \varphi)$

2.3 A Methodological Context

Using an example taken from the Dutch procedures for crisis management, we elaborate how changes in the environment affect organizations. The modelling phase is conducted according to the OperA methodology [2] using the OperettA tool [14] for the depicted diagrams. We describe how, due to the changes in the environment, the organization is forced to reorganize and what changes to its norms are required. The organizational and social structure diagram of Figure 2 represents the crisis management organization. This diagram shows the responsibilities and commitments of each participant, e.g., in case of an emergency call, the Fire_Station is dependent on Emergency_Call_Center to be informed about the disaster location. For the sake of simplicity, we consider that

Firefighter_Team sets up a strategic intervention (to achieve its primary objective extinguish fire) on the results of two evaluation criteria: *damage evaluation* and *fire evaluation*. From the former criterion, Firefighter_Team checks how many wounded there are in order to come up with information about the necessity or not to ask for ambulance_service. Moreover, the Firefighter_Team checks if the damage involves building structures that may collapse, causing obstacles and risks for drivers on the roads, e.g., this may also imply police intervention to deviate traffic in safe directions. From the fire evaluation criterion, Firefighter_Team can decide whether it is the case or not to ask Fire_Station for a Firefighting_Truck intervention.

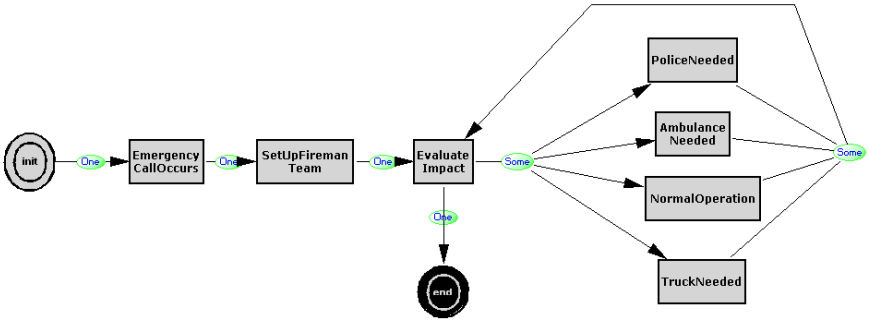


Fig. 3. Interaction Structure diagram: Fire Station example

The social structure in figure 2 describes organizational objectives and social responsibilities between organizational roles, but it gives no indication of how organizational goals can be achieved and interactions planned. This is represented in the interaction structure diagrams, which provide a partial order on how to fulfil responsibilities and to achieve objectives as depicted in Figure 3. The sequence of scenes depicted indicates how to cope with an emergency call, e.g., the Fire_Station's responsibility to build up a fire brigade each time an emergency call occurs (scene SetUpFiremanTeam) and the Firefighter_Team's responsibility of reporting on and taking decisions about the running accident (scene EvaluateImpact). OperA enables designers to further detail each scene in terms of involved organizational roles along their objectives and responsibilities, norms that have to be considered to correctly achieve objectives, and expected results at the end of the scene execution.

3 Forms of Organizational Structure Adaptation

In order to keep effective, organizations must strive to maintain a good fit in a changing environment. Changes in the environment lead to alterations on the effectiveness of the organization and therefore to the need to reorganize, or in the least, to the need to consider the consequences of that change to the organization's effectiveness and efficiency. On the other hand, organizations are active entities, capable not only of adapting to the environment but also of changing that environment. This means that organizations

are in state of, to a certain degree, altering environment conditions to meet their aims and requirements, which leads to the question of how and why reorganization decisions should be reached. The flexibility of an organization is defined as the combination of the changeability of an organizational characteristic (structure, technology, culture) and the capabilities of management to change that characteristic [9].

In Organizational Theory, the concept of *adaptation* can mean different things, ranging from strategic choice to environmental determinism. Strategic choice refers to the planned pursuit of ends based on a rational assessment of available means and conditions, resulting on a explicit decision to change the organization. Deterministic views on adaptation, on the other hand, explain organizational change as (involuntary) response to environmental requirements. In this paper, we treat adaptation as a *design* issue that requires an (explicit) action resulting in the modification of some organizational characteristics. Such decisions can be of two kinds: proactive, preparing the organization in advance for an expected future, and reactive, making adjustments after the environment has changed [4].

In terms of the formal model of organizations introduced in the following section, changes are represented as (temporal) transitions between two different worlds. Given a world $w \in W$, many different events may happen that change some proposition in that world resulting in a different world (the relation T between two worlds represents this). Because not all parameters in w are controllable by the organization, the current state of the organization is not necessarily, and in fact in most cases not, completely controlled by the organization itself. That is, changes are not always an effect of (planned) organizational activity. We distinguish between exogenous and endogenous change. In the exogenous situation, changes occur outside the control, and independently of the actions, of the agents in the organization, whereas that in the endogenous case, changes that are result of activity explicitly taken by agents in the organization.

Contingency theory [6] states that there is no one best way to organize or structure the organization, but not all structures are equally effective, that is, organizational structure is one determinant of organizational performance. Performance of the organization can be seen as the measure to which its objectives are achieved at a certain moment. Because environments evolve, performance will vary. Many organizational studies are therefore concerned with the evaluation of performance, identifying triggers for change and determine the influence of environment change in the organizational performance and indicating directions to improve performance.

In summary, reorganization consists basically of two activities. Firstly, the formal representation and evaluation of current organizational state and its 'distance' to desired state, and, secondly, the formalization of reorganization strategies, that is, the purposeful change of organizational constituents (structure, agent population, objectives) in order to make a path to desired state possible and efficient.

4 Towards a Formal Interpretation

A formal model of adaptation enables the (offline) analysis of organization models in terms of performance and utility, and can also be used to provide (runtime) decision-making support. In the following, we describe the main aspects of this formal model.

The section 4.1 the basic organization model is introduced and in section 4.2 this model is extended to deal with reorganization issues.

4.1 Formal Organization Model

Based on the formal model [3,5], outlined in Section 2.2, we provide a set of formal ingredients required to define a formal interpretation for the adaptation of organizational structures, as informally described in previous sections.

For the sake of simplicity, let us consider that each of the roles of the *fire station* organization (O), that is, Fire_Station (fs), Firefighter_Team (ft), Emergency_Call_Centre (ecc), Firefighting_Truck (fft), Police_Station (ps), and First_Aid_Station (fas), is played by a single actor, or *role enacting agent*. Moreover, we assume that the organization has the (main) objectives, handle_emergency_call (hec), determine emergency_location (el), form_a_brigade (fb), extinguish fire (ef), and sub-objectives reach_the_place (rp), how_to_react (hr), level_of_emergency (le), deal_with_big_fire (dbf), deal_with_traffic (dt), and ask for ambulance_service (as). Therefore,

$O = \{A_O, \leq_O, D_O, S^0_O\}$, where
$A = \{fs, ft, ecc, fft, ps, fas\}$
$\leq_O = \{fs \leq_O ecc, fs \leq_O ft, ft \leq_O fft, ft \leq_O ps, ft \leq_O fas\}$
$D_O = \{hec\} = \{(el \wedge fb \wedge ef)\}$

The sets A_O, and $\leq_O$ provide a formal interpretation of the roles and dependencies depicted in Figure 2.

D_O provides the formal representation of the objectives of the organization. We use the logical connectives $\wedge$ (and) and $\vee$ (or) to model sub-objective decomposition, describing possible ways for objective achievement. Informally, D_O means that an emergency call is handled (hec) if location has been identified (el), a rescue team has been formed (fb) and the fire extinguished (ef). It is important to stress here that objectives are not actions, but describe desired states of affairs to be achieved. Objectives can be further decomposed into sub-objectives, e.g. extinguish fire $ef = (rp \wedge le \wedge hr)$, namely, the fire brigade has to reach the place (rp), establish the accident severity (le) and then determine the appropriate reaction (hr). Moreover, (sub)objectives can describe alternative ways of achievement, e.g. $hr = \{(ef) \vee (as \wedge dbf) \vee (as \wedge dt \wedge dbf)\}$, meaning that appropriate ways for a firefighter team to react to a fire are to extinguish it itself, to ask for ambulance and fire truck assistance, or to ask for ambulance, fire truck and traffic coordination assistance.

The initial set S^0_O describes the capabilities and responsibilities of agents in A_O. For clarity of reading, we split S^0_O into two sub-sets $S1$ and $S2$ for capabilities and responsibilities, respectively.

$S^0_O = (S1 \cup S2)$, where:
$S1 = \{C_{ecc}el, C_{fs}fb, C_{ft}rp, C_{ft}ef, C_{ft}hr, C_{ft}le, C_{fft}dbf, C_{ps}dt, C_{fas}as\}$
$S2 = \{R_{fs}hec\}$

Given this initial state of affairs, different strategies to realize the organizational objective handle_emergency_call (hec) are possible, for example:

s_0: $R_{fs}hec$	(organizational fact)
s_1: $R_{ecc}el \wedge R_{fs}fb$	(properties of R and $fs \leq_O ecc$)
s_2: $R_{ft}rp \wedge R_{ft}le \wedge R_{ft}R_{fft}dbf$	(properties of R and $fs \leq_O ft$)
s_i: ...	
s_j: $E_{ft}le \wedge ...$	(capabilities of ft)
s_3: ...	
s_n: $le \wedge fb \wedge ef \rightarrow hec$	

That is, agent fs that enacts the role Fire_Station is, by the definition of the organization, responsible for the organization objective handle_emergency_call (hec), and has delegation power over the agents enacting organizational roles Fire_Station (fs) and Emergency_Call_Centre (ecc) as depicted in Figure 3 and formalized by the relation $\leq_O$ above. Hence, it can transfer the responsibility for organizational sub-objectives to these agents (cf. state s_1). In the same way fs can transfer responsibility for sub-objectives to ft. Finally, agents can act on their capabilities so that the organizational objective is achieved.

The formal interpretation above describes the intended meaning defined in the *Social Dependency* diagram of Figure 2 and the model annotations provided for each scene. As discussed before, social structure models do not contain information about ordering of actions, which is described in interaction structure models. The formal interpretation of interaction structure information requires the use of branching time logics. For simplicity sake, we do not consider here the interpretation of the *Interaction Structure* diagram of Figure 3.

It is worth noticing that the provided formal model reflects the main idea of the OperA methodology about the desire to preserve a good trade-off between the organizational specification and the service system flexibility. In other words, despite agents having to adhere to the organizational specification, these agents still have enough autonomy to achieve organizational objectives in convenient ways. In fact, the formal specification reflects the same aim, namely, an organizational structure may bring about different strategies (e.g., s_1 and s_2), while guaranteeing the achievement of the same set of objectives. According to each of the causes of context change described by the examples in the previous sections, we provide a formal interpretation of the required adaptation process that should be followed by the organization.

4.2 Reorganization Analysis

In the following, we describe the formalization of the different reorganization possibilities discussed in section 3. The reorganization activities described below are depicted in Figure 4.

Adaptation by re-considering the needs of the context. Imagine that the accident described in section 2.3 increases in severity. As a consequence, there is a need to

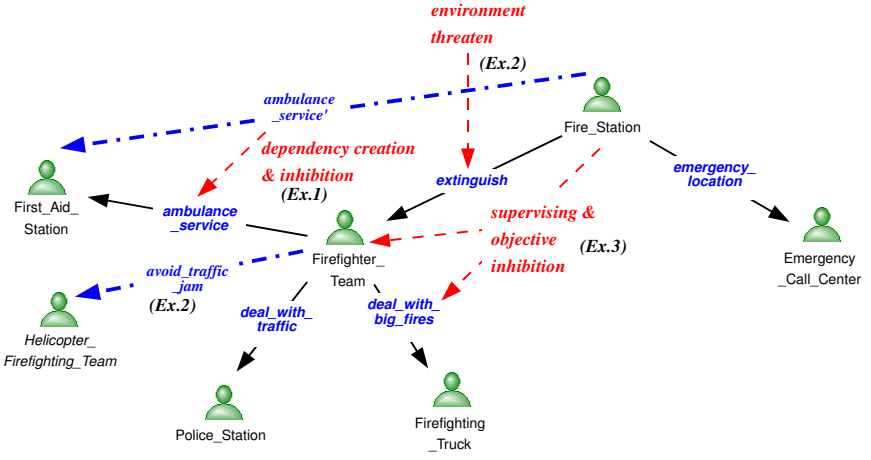


Fig. 4. Structural adaptation, according to scenarios of Example 1 (Ex.1), Example 2 (Ex.2), and Example 3 (Ex.3)

change the coordination and responsibility functions. As the accident involves several (autonomous) teams of firefighters, the responsibility of calling ambulance services is placed with the Fire_Station because the Fire_Station can better coordinate these services over different accident locations.

This context change is modeled by the creation of a dependency between Fire_Station and First_Aid_Station, as depicted in Figure 4 and removing the current dependency between Firefighter_Team and First_Aid_Station. The formal reorganization steps that are required for the organization to adapt its structure are the following:

- $R_{fs}struct^{-}(O, ft \leq_O fas)$, to eliminate the old dependency;
- $R_{fs}struct^{+}(O, fs \leq_O fas)$, to add the new dependency;

Adaptation by sensing the environment. Let's now consider that while Firefighter_Team is en route to the accident location (L), a huge traffic jam blocks the quickest access route and no alternative routes are possible; hence, the objective extinguish fire is in jeopardy. Moreover, no other cars and trucks are available but there is a helicopter available and ready for take off.

This change in the environment (symptom) could cause a failure of the objective extinguish fire if no counter measures (enforcement mechanisms) have been considered in advance. When no other alternatives are available, the enforcement has to trigger a new objective, namely, asking for the authority to use the helicopter. Figure 4 shows a possible modification to adapt to the change, in this case, Firefighter_Team collaborates with a new role Helicopter_Firefighting_Team.

This is the case where environmental changes may lead to potential service system failures. Hence, the model should include countermeasures, which imply changes in the initial organizational structure, as illustrated in Figure 4. Formally, the steps required for the organization to adapt its structure are the following:

- $R_{fs}staff^{+}(O, hft)$, add a new agent able to enact the role Helicopter_Firefighter_Team (hft), this also introduces the responsibility for the Fire_Station to make a helicopter available, e.g., during emergency conditions;
- $R_{fs}struct^{+}(O, ft \leq_O hft)$, this defines the possibility for the Firefighter_Team to call on Helicopter_Firefighter_Team (but only during emergency conditions).
- $R_{ft}strateg^{+}(O, atj)$, where atj is the objective avoid_traffic_jam, and $atj = ((symptom_1 \wedge symptom_2) \rightarrow ah)$. This formalizes the presence of emergency conditions, namely, the possibility for the agent ft to ask agent hft for help (ah - ask_for_help) but only if the two environment symptoms occur.

Adaptation by supervising system services invocation. Finally, assume that Fire_Station has to supervise Firefighter_Team movements. Thanks to a GPS locator system, Fire_Station knows whether the Firefighter_Team's achievement of the objective evaluate_fire respects the rule specifying that the accident evaluation must be done after arriving at the accident location. Therefore, a faulty request from the Firefighter_Team for having support from Firefighting_Truck can be suppressed. Organizational knowledge has a key role in the whole framework, where role and objective concepts need to be properly grounded in specific system functionalities (e.g., agent capabilities). The software agent (or service system) that enacts the role of Firefighter_Team has to adhere to the organizational norms, which regulate the service invocation. These norms have been provided by the designer as a part of the organizational model. Nevertheless, the model should provide an organizational actor with the responsibility to supervise other actor activities, namely, in Figure 4, Fire_Station supervises the fulfillment of objective deal_with_big_fires.

Service system level should take into account policies specified at the organizational level. Fire_Station agent should be able to supervise the agent Firefighter_Team, in particular when it invokes the service that brings about the objective deal_with_big_fire (dbf) achievement. In fact, this may be the cause of failure at the service level, e.g., due to a wrong pre-condition in service invocation. Therefore, within our formal specification, we need to define the agent's ability to supervise specific activities of others agents, as follows.

Definition 5 (Supervising Ability). *Given an organization $O = \{A_O, \leq_O, D_O, S_O\}$ and two agents $a, b \in A_O$, the supervising ability of an agent a with respect to another agent b to realize φ is defined as:*

$$SA_{(a,b)}\varphi \equiv R_a H_b \varphi \rightarrow \Diamond((H_b\varphi \wedge \bigcirc\neg\varphi) \rightarrow R_a\varphi).$$

This means that agent a is responsible that agent b attempts to realize φ. However, agent a is directly responsible for the achievement of φ every time b's attempt to realize φ fails. Notice that, contrary to other context changes where the change depends on a variable that models some domain entities, here, the change depends on how the activity of another agent is fulfilled. Turning to our example, to effectively deal with the context change, we need a modification within the initial state $S2$, adding the new state: $SA_{(fs,ft)}ef$, where $ef = (rp \wedge le \wedge hr)$ is the previous objective extinguish fire.

This modification guarantees Fire_Station (fs) to be informed every time Firefighter_Team (ft) tries to deal with le or hr before achieving reach_the_place (rp).

Notice that, as given by Definition 5, Fire_Station becomes directly responsible of ef in the case Firefighter_Team does not succeed to achieve ef. Hence, due to the definition of responsibility given in Section 2.2, fs can attempt itself to deal with ef or can attempt to make ft responsible of ef. In the latter case, ft can attempt to realize ef again.

5 Related Work

A-LTL [19] is an extension of the well known LTL logic. Their approach seems very suitable (at requirements level) to describe the expected program behavior of an adaptive program (e.g., a software agent) in order to enable for checking consistency in temporal logic specification. We share their idea that the use of temporal logic is a good way to allow for model checking or other formal verification in the context of adaptive systems. However, [19] considers (adaptive) programs as state machines, characterizing the adaptation as a program behavior change during its execution. Our work allows a more fine-grained requirements specification for agents involved in an organization. That is, we formalize adaptation (by the use of CTL temporal logic) as changes in the organization structures in terms of modifications between agent relationships and responsibilities, as it actually happens in human organizations.

There exist many different agent-oriented software engineering methodologies [11]. Only a few of them (e.g., see [18,15]) have emphasized the importance of organizational and social abstractions to shape and drive the system architectural and functional requirements. In [15,13], the Tropos methodology has been enriched to effectively deal with the development of adaptive systems, allowing designers to endow agents with adaptation abilities/features without focusing on agents societies. Our proposed framework thus complements the Tropos extension very nicely.

In [10] a good survey about self-organizing mechanisms inspired by social behaviors have been described. The authors refer to several kinds of self-organizing behaviors, which come from studies of insects and human societies as well as business and economic organizations, as an important source of inspiration for the development of adaptive systems. In the same line as the OperA methodology, the authors claim the need of socially inspired computing metaphors as a new paradigm for adaptive systems development.

6 Conclusions and Future Work

In this paper, we have studied forms of organizational adaptation to cope with different types of context change. The contribution of this paper is twofold: on one side, we studied and described adaptation within organizational structures by exploiting the OperettA design tool to produce the related diagrams. On the other side, we provided a well defined formal model based on the CTL temporal logic to characterize the adaptation process also from a dynamic dimension. Moreover, the paper shows that these two forms of requirements specification complement one another. Worth noticing, the formal specification gives the possibility to model different degrees of uncertainty regarding the way an agent has for achieving an objective (state of affair); hence, this

specification better reflects the behavior of (human) organizational roles that an agent from time to time may enact. The ideas of this paper have been illustrated by examples, within a simplified crisis management scenario suitable to emphasize design issues of organizations that operate in a real dynamic complex environment.

As future work, we will extend the OperA design methodology to allow designers to evaluate different organizational structures with respect to context changes. Moreover, the CTL formal specification allows us to envisage forms of analysis of formal properties (e.g., model checking).

Acknowledgements

This work has been performed in the framework of the FP7 project ALIVE IST-215890, which is funded by the European Community. The author(s) would like to acknowledge the contributions of his (their) colleagues from ALIVE Consortium (http://www.ist-alive.eu).

References

1. Aldewereld, H., Penserini, L., Dignum, F., Dignum, V.: Regulating Organizations: The ALIVE Approach. In: Workshop on Regulations Modelling and Deployment (ReMoD 2008), @CAISE 2008, Montpellier, France (2008)
2. Dignum, V.: A Model for Organizational Interaction: based on Agents, founded in Logic. PhD thesis, Universiteit Utrecht (2004)
3. Dignum, V., Dignum, F.: A logic for agent organization. In: Formal Approaches to Multi-Agent Systems (FAMAS@Agents 2007), Durham (2007)
4. Dignum, V., Dignum, F., Sonenberg, L.: Towards dynamic organization of agent societies. In: Vouros, G. (ed.) Workshop on Coordination in Emergent Agent Societies, ECAI 2004, pp. 70–78 (2004)
5. Dignum, V., Tick, C.: Agent-based Analysis of Organizations: Performance and Adaptation. In: IEEE/WIC/ACM Int. Conf. on Intelligent Agent Technology (IAT 2007), California, USA. IEEE CS Press, Los Alamitos (2007)
6. Donaldson, L.: The Contingency Theory of Organizations. Sage, Thousand Oaks (2001)
7. Emerson, E.: Temporal and modal logic. In: van Leeuwen, J. (ed.) Handbook of Theoretical Computer Science, vol. B, pp. 955–1072. MIT Press, Cambridge (1990)
8. Ganek, A.G., Corbi, T.A.: The dawning of the autonomic computing era. IBM Systems Journal 42(1), 5–18 (2003)
9. Gazendam, H., Simons, J.L.: An analysis of the concept of equilibrium in organization theory. In: Proceedings of the Computational and Mathematical Organization Theory Workshop (1998)
10. Hassas, S., Marzo-Serugendo, G.D., Karageorgos, A., Castelfranchi, C.: Self-organising mechanisms from social and business/economics approaches. Informatica 30(1), 63–71 (2006)
11. Henderson-Sellers, B., Giorgini, P. (eds.): Agent-Oriented Methodologies. Idea Group Inc., USA (2005)
12. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. Computer 36(1), 41–50 (2003)

13. Morandini, M., Penserini, L., Perini, A.: Operational Semantics of Goal Models in Adaptive Agents. In: Proceedings of the Eighth International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2009). ACM, New York (2009)
14. Okouya, D.M., Dignum, V.: A prototype tool for the design, analysis and development of multi-agent organizations. In: DEMO Session: Proc. of the 7th Int. Joint Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2008). ACM, New York (2008)
15. Penserini, L., Perini, A., Susi, A., Mylopoulos, J.: High Variability Design for Software Agents: Extending Tropos. ACM Transactions on Autonomous and Adaptive Systems (TAAS) 2(4) (2007)
16. Pörn, I.: Some basic concepts of action. In: Stenlund, S. (ed.) Logical Theory and Semantical Analysis. Reidel, Dordrecht (1974)
17. So, Y., Durfee, E.H.: Designing organizations for computational agents. In: Simulating Organizations: Computational Models of Institutions and Groups, pp. 47–64. MIT Press, Cambridge (1998)
18. Zambonelli, F., Jennings, N.R., Wooldridge, M.: Developing multiagent systems: The gaia methodology. ACM Transactions on Software Engineering and Methodology 12(3), 317–370 (2003)
19. Zhang, J., Cheng, B.H.C.: Using Temporal Logic to Specify Adaptive Program Semantics. Journal of Systems and Software (JSS) 79(10), 1361–1369 (2006)

GORMAS: An Organizational-Oriented Methodological Guideline for Open MAS

Estefanía Argente, Vicent Botti, and Vicente Julian

Departamento de Sistemas Informáticos y Computación
Universidad Politécnica de Valencia
Camino de Vera s/n. 46022, Valencia, Spain
{eargente,vbotti,vinglada}@dsic.upv.es

Abstract. An Organizational-Oriented Methodological Guideline for designing Virtual Organizations, focused on Service-Oriented Open Multi-Agent Systems, is presented in this paper. This guideline covers the requirement analysis, the structure design and the organization-dynamics design steps, in which software designers mainly specify the services that the system under study is requested to offer, the internal structure of this system and the norms that control its behavior, taking into account the specific features of open multi-agent systems.

Keywords: virtual organizations, multi-agent systems, methodology.

1 Introduction

In the last years, there have appeared different methods for designing open multi-agent systems, in which heterogeneous agents with interested or selfish behaviors might participate inside. These methods take an *Organization-Centered Multi-Agent Systems (OCMAS)* [18] approach, so then developers focus on the organizational aspects of the society of agents, guiding the process of the system development by means of organizations, norms, roles, etc. Relevant examples of these methods are Agent-Group-Role (AGR) [18], Tropos [9], MOISE [22], OMNI [37] (based on the E-Institutions [17] approach), PASSI [10], SODA [30] and INGENIAS [35].

A key concept in OCMAS methodologies is the *Virtual Organization* (VO), which represents a set of single entities and institutions that need to coordinate resources and services across institutional boundaries [12, 3]. Thus, they are open systems formed by the grouping and collaboration of heterogeneous entities (that may be designed by different teams) and there is a separation between form and function that requires defining how a behavior will take place. They have been successfully employed as a paradigm for developing agent systems [4, 10]. Organizations allow modeling systems at a high level of abstraction. They include the integration of organizational and individual perspectives and also the dynamic adaptation of models to organizational and environmental changes [7] by forming groups with visibility boundaries [10].

M.-P. Gleizes and J.J. Gomez-Sanz (Eds.): AOSE 2009, LNCS 6038, pp. 32–47, 2011.

From an analysis of these OCMAS methodologies [4], several critical needs have been identified: (i) a clear characterization of the VO focused not only on describing its structure by means of roles and groups, but also on how this organization its related with its environment and how external entities are promoted to enter inside and offer their services or make use of the organization services; (ii) the employment of several design patterns of different types of organizational structures, which describe the intrinsic relationships between the system entities and can be lately reused in different problems; (iii) a normative regulation of the employment of services that also promotes agents to work in a cooperative way; and (iv) different guidelines for integrating the current OCMAS methodologies in a complete MAS development process.

Regarding services, both Multi-Agent Systems and Service-Oriented Computing Techniques[1] try to deal with the same kind of environments formed by loose-coupled, flexible, persistent and distributed tasks [28]. Web services might be a valid solution when point-to-point integration of static-bound services is needed, but they are clearly not good enough for working in a changing environment, in which new services appear, have to be discovered and composed or adapted to different ontologies. The nature of agents, as intelligent and flexible entities with auto-organizational capabilities, facilitates automatic service discovery and composition. By integrating these two technologies it is possible to model autonomous and heterogeneous computational entities in dynamic and open environments [27].

Following this integrating approach, the THOMAS proposal [8] provides a multi-agent framework for the development of VOs with a service-oriented style. THOMAS [8] is an open agent platform that employs a service-based approach as the basic building blocks for creating a suitable platform for intelligent agents grouped in VOs. It feeds on the FIPA architecture but extends its main components (i.e. AMS and DF) into an Organization Management System (OMS) and a Service Facilitator (SF), respectively. The SF [36] is a service manager that registers services provided by external entities and facilitates service discovery for potential clients. The OMS [13] is responsible for the management of VOs, taking control of their underlying structure, the roles played by the agents inside the organization and the norms that rule the system behavior.

An OCMAS method should be employed to facilitate the identification of VOs and services of a specific problem to be implemented organization-oriented platforms like THOMAS. In this paper, a methodological guideline for MAS designing based on the Organization Theory [14,19] and the Service-Oriented approach [16] has been defined, named GORMAS, which allows detailing all services offered and required by a Virtual Organization, its internal structure, functionality, normative restrictions and its environment.

Next, GORMAS, the organization-centered MAS methodological guideline defined in this work is presented. In section 3, a brief discussion of the related work is shown. Finally, conclusions are included in section 4.

[1] `http://www.oasis-open.org`

2 GORMAS

GORMAS (**G**uidelines for **OR**ganizational **M**ulti-**A**gent **S**ystems) defines a set of activities for the analysis and design of Virtual Organizations, including the design of their organizational structure and their dynamics. With this method, all services offered and required by the Virtual Organization are clearly defined, as well as its internal structure and the norms that govern its behavior.

GORMAS is based on a specific method for designing human organizations [31,32], which consists of diverse phases for analysis and design. These phases have been appropriately transformed to the MAS field, this way to catch all the requirements of the design of an organization from the agents' perspective. Thus, the methodological guidelines proposed in GORMAS cover the typical requirement analysis, architectural and detailed designs of many relevant OOMAS methodologies, but it also includes a deeper analysis of the system as an open organization that provides and offers services to its environment.

The proposed guideline allows being integrated into a development process of complete software, which may include the phases of analysis, design, implementation, installation and maintenance of MAS. In this paper, we mainly focus on the analysis and design processes (Figure 1a), which are split into: mission and service analysis steps (analysis phase); and organizational and organization-dynamics design steps (design phase). Implementation is carried out in the THOMAS framework [8] which mostly covers the organization software components that are required, such as organizational unit life-cycle management, service searching and composition and norm management.

GORMAS adopts a Virtual Organization Model [11], formalized in a set of six models [5]: (i) *organizational*, which describes the entities of the system (agents, organizational units, roles, norms, resources, applications) and how they are related between them (social relationships); (ii) *activity*, which details the specific functionality of the system, on the basis of services, tasks and goals; (iii) *interaction*, which defines the interactions of the system, activated by the pursuit of goals and the execution of services; (iv) *environment*, which describes the resources and applications of the system, the agent perceptions and actions on their environment and the invocation of services through their ports; (v) *agent*, which describes the concrete agents and their responsibilities; and (vi) *normative*, which details the norms of the organization and the normative goals that agents must follow, including sanctions and rewards.

This Virtual Organization Model employs the *Organizational Unit* (OU) concept to represent an agent organization. An OU is composed of a group of entities (agents or OUs) that carry out some specific and differentiated tasks, following a pattern of cooperation and communication controlled by norms [6,5]. The OU is seen as a single entity at analysis and design phases, thus it can pursue goals, offer and request services, publish its requirements of services for allowing external agents to enter inside, and even play a specific role inside other units.

Due to lack of space, a general view of the different activities that integrate the proposed methodological guideline is only described in this paper. These

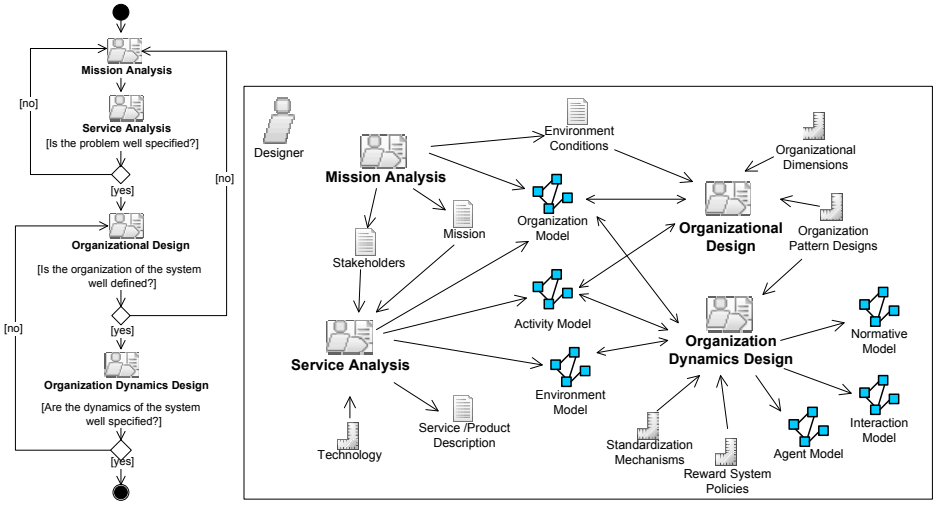


Fig. 1. a) GORMAS Activity Diagram; b) GORMAS Activity Detail Diagram

activities are defined with SPEM 2.0[2], which is an OMG standard meta-model for formally defining software and system development processes. Moreover, GORMAS activities include several supporting documents and templates for enabling the identification and description of the elements of the system [1], such as templates for identifying the system functionality and describing service conditions, consumer/producer goals and quality requirements. Figure 1b details all specific guidances, models and work products that each activity needs or produces.

2.1 Mission Analysis

This first activity (Figure 2a) implies the analysis of the system requirements, identifying the use cases, the stakeholders and the global goals of the system. More concretely, it is defined:

- The global goals of the system (mission).
- The services and products that the system provides to other entities.
- The stakeholders with whom the system contacts (clients or suppliers of resources/services), describing their needs and requirements.
- The conditions of the environment or context in which the organization exists (i.e. complexity, diversity, etc.).

As a result, a diagram of the *organizational model* is drawn, detailing the products and services offered by the system, the global goals (mission) pursued, the stakeholders and the existing links between them, the system results as well as the resources or services needed.

[2] `http://www.omg.org/spc/SPEM/2.0/PDF`

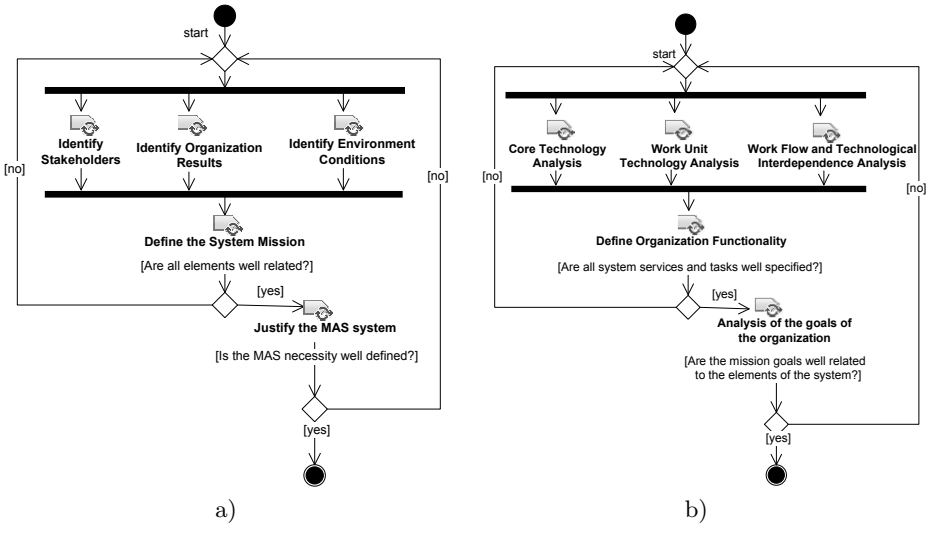


Fig. 2. Activity Diagrams of: a) *Mission Analysis* step; b) *Service Analysis* step

2.2 Service Analysis

In this activity (Figure 2b), the services offered by the organization to its clients are specified, as well as how these services behave (inputs/outputs description and internal tasks) and which are the relationships (interdependencies) between these services. Furthermore, goals associated with services are detailed. More specifically, it is specified:

- The type of products and services that the system offers to or consumes from its stakeholders.
- The tasks related to the production of the products and services, defining the steps necessary for obtaining them, their relationships and interdependences between the different services and tasks.
- The goals related with the achievement of these products and services.
- The resources and applications needed for offering the system functionality.
- The roles related with the stakeholders, on the basis of the type of services or tasks that they provide or consume.

Taking the Organization Theory [14,19] as a basis, the three existing types of technology are considered [26]: (i) the *Core Technology*, which refers to the whole organization; (ii) the *Work Unit or Departmental Technology*, which contemplates the diversity and complexity of the different organizational tasks, identifying the existing workflows; and (iii) *Work Flow and Technological Interdependence*, that defines the interdependent relations originated as a result of the workflow between the units of the organization.

Thus, in the *Core Technology Analysis* activity, the type of products and services that the system offers to or consumes from its stakeholdres are determined,

as well as in which measure does any client have influence in the process of production and the final aspect of the product; on in which measure services are related between them and with regard to clients. In the *Work Unit Technology Analysis* activity, an instance of the activity model for each service is generated, detailing its profile (service inputs, outputs, preconditions and postconditions). In addition, a first decomposition of services in sequence of steps or tasks is carried out. Finally, in the *Work Flow and Technological Interdependes* activity, the existing relations between the units of the organization are specified in order to reach the organizational goals. In this way, the flows of tasks that turn out to be totally independent between them are firstly analyzed. Thus, these tasks that are not directly related between them and, therefore, they will be able to be executed simultaneously are specified. Next, the sequence of steps needed between the dependent tasks is also identified.

It should be pointed out that typical business workflows which represent organization dynamics are modeled in GORMAS by means of services. Thus, a service can be defined as a composition of tasks, similarly to a workflow in INGENIAS[25], where a workflow is considered as a chain of tasks, connected with *WFConnects* relationships.

In the *Analysis of the goals of the organization* activity, the mission is decomposed into functional goals, which are related to the system entities and identified services. The *functional goals* [26,38] or *soft-goals* [23] represent the specific actions of the units of the organization and their expected results. Normally they are based on some of the following types of goals: satisfaction of the client and/or of other groups of interest; continuous improvement of the processes, products and services of the organization; cooperation between all the members, active participation, commitment with their tasks, etc.

As a result of this phase, the diagrams of the organizational and the activity models are generated. More concretely, in the *organizational model*, both resources and applications of the system are identified, and also the entities representing the clients or providers of the system that are required to participate inside. Moreover, the services required and offered by the system are identified, as well as the roles that provide or make use of these services. For each service, a diagram of the *activity model* is generated, detailing its profile and its tasks. Finally, in the *activity model*, the mission is split into functional goals, which are related to the system entities and their services.

2.3 Organizational Design

In this development step (Figure 3a), the structure most adapted for the Virtual Organization is selected. This structure will determine the relationships and pre-established restrictions that exist between the elements of the system, based on specific dimensions of the organization [29,38], which impose certain requirements on the types of work, on the structure of the system and on the interdependences between tasks. These organizational dimensions are:

- *Departmentalization*, which details the motivation of work group formation, i.e. functional (on the basis of knowledge, skills or processes) or divisional (on the basis of the characteristics of the market, clients, products or services).
- *Specialization*, which indicates the degree of task dividing, based on the quantity and diversity of tasks(horizontal job specialization) and the control exercised on them (vertical job specialization).
- *Decision making*, that determines the degree of centralization of the organization, i.e. the degree of concentration of authority and capture of decisions.
- *Formalization*, which specifies the degree in which the tasks and positions are standardized, by means of norms and rules of behavior.
- *Coordination Mechanism*, which indicates how individuals can coordinate their tasks, minimizing their interactions and maximizing their efficiency, using mutual adjustment, direct supervision or standardization.

These organizational dimensions are employed for determining the most suitable structure for the system specifications. Thus, it is carried out:

- The analysis of the organizational dimensions, which allows specifying the *functionality granularity* of the service [24], by means of grouping tasks and services together (defining more complex services) and assigning them to specific roles; and identifying general restrictions on the coordination and cooperation behavior of the members of the organization.
- The selection of the structure of organization most interesting for the system.
- The adaption of the selected structure to the problem under study, using specific design patterns.

For the structure selection, a decision-tree has been elaborated (Figure 3b), that enables the system designer to identify which is the structure that better adjusts to the conditions imposed by the organizational dimensions. This decision-tree can be applied to the system as a whole or to each of its OUs, so then enabling structure combinations.

A set of design patterns of different structures has been defined which include simple hierarchy, team, flat structure, bureaucracy, matrix, federation, coalition and congregation structures [4]. These patterns describe their intrinsic structural roles, their social relationships, as well as their typical functionality. According to these design patterns, the diagrams of the organizational and activity models are updated, integrating these intrinsic roles, relationships and functionality.

2.4 Organization Dynamics Design

In this activity (Figure 4a), the detailed design of the system is carried out, which implies the following activities: the design of the information-decision processes; the design of the system dynamics as an open system; the design of the measuring, evaluation and control methods; the definition of a suitable reward system; and finally, the design of the system agents, describing them by means of diagrams of the agent model. These activities are detailed as follows.

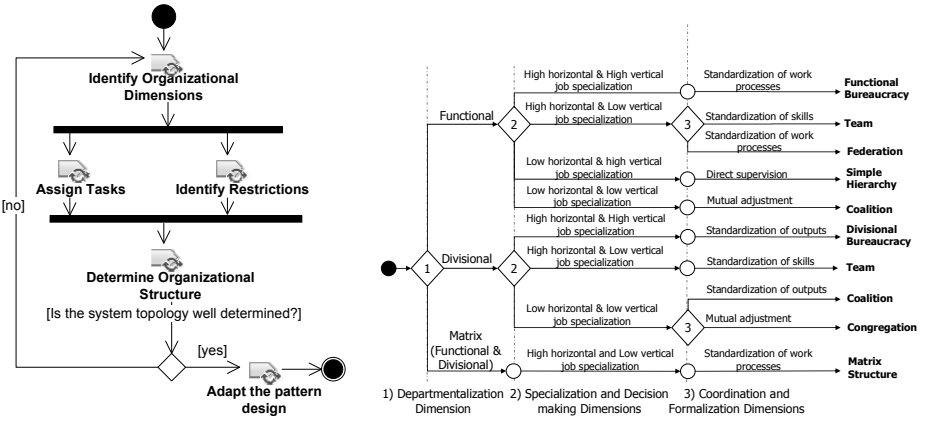


Fig. 3. a) Activity Diagram of the *Organizational Design* step; b) Organizational Structure decision-tree

Design of Information-Decision Processes. The flows of information and adoption of decisions are described in order to determine how the information is processed and how agents work for obtaining the expected results (Figure 4b). More concretely:

- The concrete functionality of services is specified, which implies:
  - Detailing services and related workflows for providing these services and splitting these workflows in concrete tasks.
  - Identifying the operative goals, that represent the specific and measurable results that the members of an OU are expected to achieve. Thus, functional goals are split into operative goals, which are lately assigned to tasks.
- The flow of information of the system is specified, which implies:
  - Checking the procedures for obtaining information of the environment, i.e. verifying whether there exists a contact point with the system for any stakeholder, using resources, applications or representative agents.
  - Defining permissions for accessing resources or applications.
  - Defining the specific interactions between agents on the basis of services and their collaboration diagrams, in which agent messages are defined.
  - Defining the ontology of the domain, whose concepts will be used in the interactions, taking into account the service inputs and outputs.

As a result, the diagrams of the *interaction model* are defined. Moreover, both environment, organization and activity model diagrams are updated.

Design of Open System Dynamics. In this activity (Figure 5a), the functionality offered by the Virtual Organization as an open system is established, which includes so much the services that must be advertised as the policies of

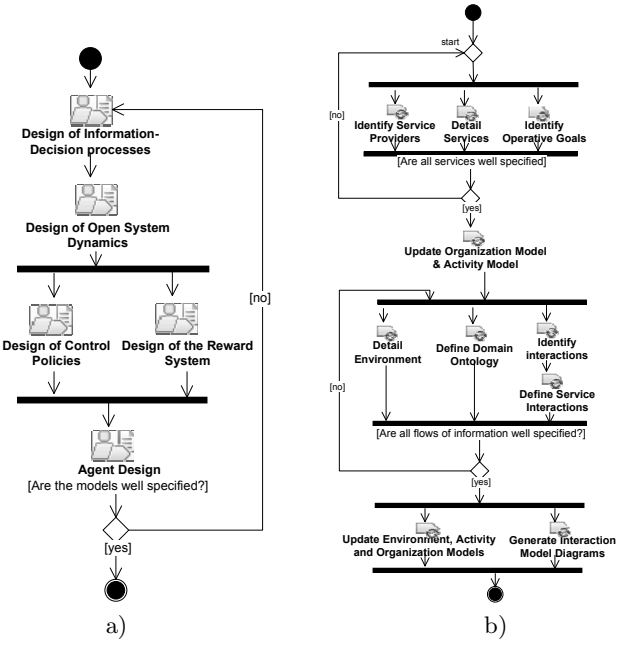


Fig. 4. Activity Diagrams of: a) *Organization Dynamics Design* step; b) *Information-Decision Design* step

role enactment. In this sense, it is specified which is the functionality that must be implemented by the internal agents of the system and which must be advertised in order to enable this functionality to be provided by external agents. Therefore, it is determined:

- The services that have to be advertised.
- The policies for role enactment, detailing the specific tasks for the *Acquire-Role* and *LeaveRole* services of each OU.
- The identification of the internal and external agents.

All roles that need a control of their behaviors require a registration process inside the OU in which they take part, so they are associated with external agents, who must request the *AcquireRole* service to play this role. On the contrary, roles like managers or supervisors are assigned to internal agents, since their functionality needs of sufficient guarantees of safety and efficiency.

Design of Control Policies. In this activity (Figure 5b), the set of norms and restrictions needed for applying the normalization dimension is defined. Three different types of standardization are considered [29]: standardization of work processes, outputs, and skills.

The *standardization of work processes* implies specifying rules for controlling: (i) invocation and execution order of services; (ii) precedence relationships

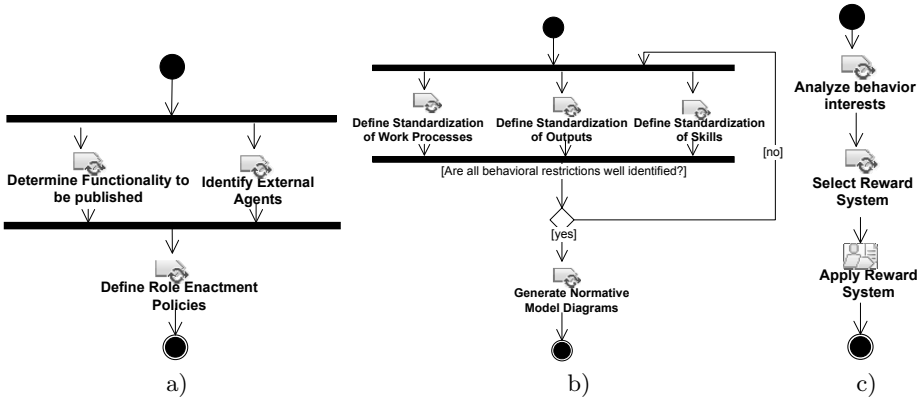


Fig. 5. Activity Diagrams of: a) *Open System Dynamics* step; b) *Design of Control Policies* step; c) *Design of the Reward System* step

between tasks; (iii) deadline and activation conditions of services; (iv) and service access to resources or applications.

The *standardization of outputs* implies specifying norms for controlling the service products, based on minimum quality requirements, perceived quality and previous established goals of productivity and performance.

Finally, the *standardization of skills* is integrated in the role concept, which indicates the knowledge and skills required for an agent when playing this role.

As a result, the diagrams of the *normative model* are generated, so the norms needed for controlling the behaviors of the members of the system are detailed. These norms can also be defined with a specific normative language [3] that imposes a deontic control over agents for requesting, providing or publishing services. In addition, a normative implementation of this normative language has been developed [12], so as to enable agents to take into consideration the existence of norms. For instance, in the THOMAS platform, the OMS component makes use of this normative implementation in order to control how services can be employed [13], i.e., when and how agents can request, provide or publish their services and the specific ones provided by the THOMAS platform.

Design of the Reward System. This activity defines the reward system needed for allowing the members of the Virtual Organization to work towards the strategy of the organization (Figure 5c). Therefore, designer proceeds to:

- Analyze the types of behavior needed to promote [20]:
  - *the willingness to join and remain* inside the system.
  - the *performance dependent on role*, so that the minimal levels of quality and quantity of work to execute are achieved.
  - the *effort on the minimal levels*, defining several measures of performance, efficiency and productivity.
  - the *cooperative behaviors* with the rest of members of the organization.

- Select the type of reward system to be used [20]:
  - *individual rewards*, which establish several measures of behavior of unit member, allowing to promote their efforts on the minimal levels.
  - *group rewards* (competitive or cooperative) [38], that establish several group measures, rewarding its members based on their specific performance inside the group.
  - *system rewards*, which distribute certain gratifications (ex. permissions, resource accesses) to all members of the system, just by belonging to this group, trying to promote the participation inside the organization.
- Apply the selected reward system in the specific problem domain.

The methodological guideline enables the selection of the type of incentive system that should be employed, but it does not detail the concrete mechanisms for implementing this system. As a result, the diagrams of the *normative model* are updated, defining new norms or adding sanctions and rewards to the existing norms, according to the selected reward system.

As summary, the methodological guide proposed for designing Virtual Organizations, based on a specific guideline for designing human organizations, provides an iterative method that consists of: (i) the analysis and design of the global goals of the organization; (ii) the analysis of its tasks and processes; (iii) the specification of the organizational dimensions; (iv) the selection of the most adapted structure for the system; (v) the identification of the processes of information and decision; (vi) the analysis of the public functionality that has to be advertised; (vii) the control of the agent behaviors and (viii) the specification of sanctions and rewards, for promoting interesting organization behaviors.

3 Related Work

Organization-centered MAS methodologies try to describe, in general, which are the main goals of the organization, its organizational structure (topology, hierarchy of roles and interactions), its dynamics (how agents enter/leave the organization, how they adopt roles, which is the agents' life-cycle and which are the social norms) and the environment of the organization. For instance, Gaia [39] considers that a specific topology for the system will force the use of several roles that depend on the selected topological pattern. Thus, it suggests employing organizational patterns in the analysis and design of the system. Tropos [9] proposes to use generic multi-agent structures based on human organizations, such as structure-in-five, pyramid, joint venture, etc. During the design phase, several social agent patterns are assigned to organizational topologies, such as brokers, matchmakers, mediators. However, social rules are not considered in Tropos and there is not any model that enables determining global rules to be applied in the organization as a whole or in multiple organizational roles.

More generally, although OCMAS methodologies are organization-centered, they normally focus on a rather reduced number of topological structures and do not take into account the human organizational designs. For example, AGR [18] only considers groups (set of agents that share common features) as a topological

structure and it assumes that groups are dynamically created during the execution of the system. In E-Institutions [17], system structure is organized by means of interaction scenes and transitions, where a scene is a communicative process between agents. These scenes are defined after analyzing role relationships, but without taking into account any human organizational topology. Therefore, it would be interesting to consider different types of organizational structures, so scene identification and development process would be easier. In SODA [33], tasks are related to roles (if they need limited resources and competencies) or to groups (if they need several competencies and access to different resources). Later, in the design phase a social model is considered, in which groups are assigned to agent societies, designed over specific coordination media that provide abstractions expressive enough to model the society interaction rules. These coordination models (such as *first come/first served* or *task decomposition*) also appear inside human societies. Therefore, although this methodology does not specifically take into account human organizational designs, it considers several human coordination mechanisms. INGENIAS [35] (based on MESSAGE[7]) specifies an organizational model in which groups, members, work flows and organizational goals are detailed. Thus, several work flows for each use case are defined, and also actors (agents and roles) participating in those work flows. Then these actors are grouped together according to their functionality or available resources that they need. Therefore, groups are identified but leaving out of account any other kind of organizational design. Finally, OperA [15] represents an important approach for modeling open multi-agent systems, since it describes the desired behavior of the society and its general structure by means of an *organizational model*, as well as details organizational dynamics using a *social model* and an *interaction model*, that describes the actual behavior of society during its execution. However, if the system is composed of several topological models or substructures, OperA does not offer any guideline to distinguish and combine these substructures between them. Moreover, OperA only takes into account three coordination models (market, hierarchy and network), but other topological systems, such as matrix organizations or chain of values, should also be taken into account.

The guideline proposed in this paper integrates both the human organization design, based on the Organization Theory, and some of the most relevant OCMAS methodologies, such as Tropos [9], Gaia [39], MOISE [22], INGENIAS [34] and the E-Institution [17] framework, as well as the Virtual Organization Model [11]. In this way, GORMAS extends the analysis requirement of Tropos with a deeper description of the mission of the system. It also includes a goal categorization into mission, functional and operative goals, similar to soft and hard goals of Tropos, but more related to the human organization design; and the selection of the organization structure, in this case based on the Organization Theory, which specifies a set of organizational dimensions that impose certain requirements on the types of work, on the structure of the system and on the task interdependences. Moreover, a set of design patterns of different structures has been defined, similarly as in Tropos or Gaia, but based on the Organization

Theory. These design patterns include hierarchy, team, bureaucracy, matrix, federation, coalition and congregation structures [4]. Furthermore, all services provided by internal entities of the system are identified, as well as all services offered by external agents which require a strict control of their final behavior. In this way, the normative features of MOISE, Gaia and E-Institutions have been integrated into GORMAS and extended with a reward system.

Additionally, GORMAS can be independently used for the analysis and design of the multiagent system, or it can be integrated in other development methods, such as INGENIAS or SODA, in order to provide them with an organizational perspective and an open system behavior. For example, GORMAS improves SODA with a justification of the MAS, an analysis of the environmental conditions, an analysis of the functionality of the system based on services, as well as a mechanism to facilitate the selection of a suitable topology for the system, based on the organizational dimensions defined in the Organization Theory. In addition, both SODA architecture and detailed designs could be reinforced with GORMAS activities, thereby making easier the description of the agent interactions, the identification of groups of agents and the description of the system rules (including sanctions and rewards). Furthermore, SODA agent societies present similarities to our Organizational Units since they both can be seen as a group of agents, but OUs also represent both resources and norms integrated in the agent society, thus being a more powerful abstraction.

4 Conclusions

This work describes GORMAS, an Organizational-Oriented methodological guideline for designing Virtual Organizations. Its main advantages are: (i) an integration of the Organization Theory for detailing the Virtual Organization Model and defining the normative and reward systems; (ii) a selection process for a suitable structure of the system organization that makes use of several design patterns of typical organizational structures; (iii) an iterative method for generating all diagrams of the Virtual Organization Model; (iv) a specification of different activities that can be integrated in a complete MAS development process; and (v) a description of the system functionality from a Service-Oriented perspective, defining not only the services offered by the organization, but also the services required inside the system that are not yet provided by the organization but have to be supplied by external agents in a regulated and controlled way. This is the most relevant advantage, since the dynamic evolution of the open MAS is considered both in analysis and design phases.

GORMAS has been applied into different application examples, such as a personalized information system [2], a water market system [21] and a tourism market system [1]; all of them considered as VOs in which external entities can participate inside and their behavior is controlled by norms. Moreover, a graphical development tool[3], named EMFGormas [21], has been defined. This tool is based on an unified meta-model for engineering large-scale open systems.

[3] `http://www.dsic.upv.es/users/ia/sma/tools/EMFgormas/index.html`

EMFGormas helps designers with the diagram model construction of the VO following GORMAS activities, verifies its consistency and generates final code to the THOMAS framework, which provides a service-oriented execution framework for supporting the development of open MAS.

Acknowledgments. This work has been partially funded by TIN2005-03395 and TIN2006-14630-C03- 01 projects of the Spanish government, FEDER funds and CONSOLIDER-INGENIO 2010 under grant CSD2007-00022.

References

1. Argente, E.: GORMAS: Guias para el desarrollo de Sistemas multi-agente abiertos basados en organizaciones. PhD thesis, Universidad Politecnica de Valencia (2008)
2. Argente, E., Botti, V., Julian, V.: Organizational-oriented methodological guidelines for designing virtual organizations. In: Omatu, S., Rocha, M.P., Bravo, J., Fernández, F., Corchado, E., Bustillo, A., Corchado, J.M. (eds.) IWANN 2009. LNCS, vol. 5518, pp. 154–162. Springer, Heidelberg (2009)
3. Argente, E., Criado, N., Botti, V., Julian, V.: Norms for agent service controlling. In: Proc. EUMAS, pp. 1–15 (2008)
4. Argente, E., Julian, V., Botti, V.: Multi-Agent System Development based on Organizations. Electron Notes Theor. Comput. Sci (ENTCS) 150(3), 55–71 (2006)
5. Argente, E., Julian, V., Botti, V.: MAS Modeling Based on Organizations. In: Luck, M., Gomez-Sanz, J.J. (eds.) AOSE 2008. LNCS, vol. 5386, pp. 1–12. Springer, Heidelberg (2009)
6. Argente, E., Palanca, J., Aranda, G., Julian, V., Botti, V.: Supporting Agent Organizations. In: Burkhard, H.-D., Lindemann, G., Verbrugge, R., Varga, L.Z. (eds.) CEEMAS 2007. LNCS (LNAI), vol. 4696, pp. 236–245. Springer, Heidelberg (2007)
7. Caire, G., Coulier, W., Garijo, F., Gomez, J., Pavon, J., Leal, F., Chainho, P., Kearney, P., Stark, J., Evans, R., Massonet, P.: Agent-oriented analysis using MESSAGE/UML. In: Wooldridge, M.J., Weiß, G., Ciancarini, P. (eds.) AOSE 2001. LNCS, vol. 2222, pp. 119–125. Springer, Heidelberg (2002)
8. Carrascosa, C., Giret, A., Julian, V., Rebollo, M., Argente, E., Botti, V.: Service Oriented MAS: An open Architecture. In: Proc. Autonomous Agents and Multiagent Systems (AAMAS), pp. 1291–1292 (2009)
9. Castro, J., Kolp, M., Mylopoulos, J.: A requirements-driven software development methodology. In: Conf. Advanced Information Systems Engineering (2001)
10. Cossentino, M.: From requirements to code with the passi methodology. In: Agent Oriented Methodologies IV, pp. 79–106 (2005)
11. Criado, N., Argente, E., Julian, V., Botti, V.: Designing Virtual Organizations. In: Proc. PAAMS 2009, vol. 55, pp. 440–449 (2009)
12. Criado, N., Julian, V., Argente, E.: Towards the implementation of a normative reasoning process. In: Proc. PAAMS, vol. 55, pp. 319–328 (2009)
13. Criado, N., Julian, V., Botti, V., Argente, E.: A Norm-based Organization Management System. In: AAMAS Workshop on Coordination, Organization, Institutions and Norms in Agent Systems (COIN), pp. 1–16 (2009)
14. Daft, R.: Organization Theory and Design. South-Western College Pub. (2003)
15. Dignum, V.: A model for organizational interaction: based on agents, founded in logic. PhD thesis, Utrecht University (2003)

16. Erl, T.: Service-Oriented Architecture: Concepts, Technology, and Design. Prentice-Hall, Englewood Cliffs (2005)
17. Esteva, M., Rodriguez, J., Sierra, C., Garcia, P., Arcos, J.: On the formal Specification of Electronic Institutions. In: Sierra, C., Dignum, F.P.M. (eds.) AgentLink 2000. LNCS (LNAI), vol. 1991, pp. 126–147. Springer, Heidelberg (2001)
18. Ferber, J., Gutknecht, O., Michel, F.: From agents to organizations: An organizational view of multi-agent systems. In: Giorgini, P., Müller, J.P., Odell, J.J. (eds.) AOSE 2003. LNCS, vol. 2935, pp. 214–230. Springer, Heidelberg (2004)
19. Fox, M.: An organizational view of distributed systems. IEEE Trans. on System, Man and Cybernetics 11, 70–80 (1981)
20. Galbraith, J.: Organization Design. Addison-Wesley, Reading (1977)
21. Garcia, E., Argente, E., Giret, A.: Service-oriented Open Multiagent Systems modeling tool. In: Yang, J.-J., Yokoo, M., Ito, T., Jin, Z., Scerri, P. (eds.) PRIMA 2009. LNCS, vol. 5925, Springer, Heidelberg (2009)
22. Gateau, B., Boissier, O., Khadraoui, D., Dubois, E.: MoiseInst: An Organizational model for specifying rights and duties of autonomous agents. In: Proc. EUMAS 2005, pp. 484–485 (2005)
23. Giunchiglia, F., Mylopoulos, J., Perini, A.: The Tropos Software Development Methodology: Processes, Models and Diagrams. In: Giunchiglia, F., Odell, J.J., Weiss, G. (eds.) AOSE 2002. LNCS, vol. 2585, pp. 162–173. Springer, Heidelberg (2003)
24. Haesen, R., Snoeck, M., Lemahieu, W., Poelmans, S.: On the Definition of Service Granularity and its Architectural Impact. In: Bellahsène, Z., Léonard, M. (eds.) CAiSE 2008. LNCS, vol. 5074, pp. 375–389. Springer, Heidelberg (2008)
25. Hidalgo, A.G., Gomez-Sanz, J.J., Pavón, J.: Workflow Modelling with INGENIAS methodology. In: Proc. 5th IEEE Int. Conf. on Industrial Informatics, vol. 2, pp. 1103–1108 (2007)
26. Hodge, B.J., Anthony, W., Gales, L.: Organization Theory: A Strategic Approach. Prentice-Hall, Englewood Cliffs (2002)
27. Huhns, M., Singh, M.: Reseach directions for service-oriented multiagent systems. IEEE Internet Computing, Service-Oriented Computing Track 9(1) (2005)
28. Huhns, M., Singh, M.: Service-oriented computing: Key concepts and principles. IEEE Internet Computing, Service-Oriented Computing Track 9(1) (2005)
29. Mintzberg, H.: Structures in fives: designing effective organizations. Prentice-Hall, Englewood Cliffs (1992)
30. Molesini, A., Omicini, A., Denti, E., Ricci, A.: SODA: A roadmap to artefacts. In: Dikenelli, O., Gleizes, M.-P., Ricci, A. (eds.) ESAW 2005. LNCS (LNAI), vol. 3963, pp. 49–62. Springer, Heidelberg (2006)
31. Moreno-Luzon, M.D., Peris, F., Gonzalez, T.: Gestión de la Calidad y Diseño de Organizaciones. Prentice Hall, Pearson Education (2001)
32. Moreno-Luzon, M.D., Peris, F., Santonja, J.: Quality management in the small and medium sized companies and strategic management. In: The Handbook of Total Quality Management, pp. 128–153 (1998)
33. Omicini, A.: SODA: Societies and Infrastructures in the Analysis and Design of Agent-Based Systems. In: Ciancarini, P., Wooldridge, M.J. (eds.) AOSE 2000. LNCS, vol. 1957, pp. 185–193. Springer, Heidelberg (2001)
34. Pavón, J., Gómez-Sanz, J.J.: Agent Oriented Software Engineering with INGENIAS. In: Mařík, V., Müller, J.P., Pěchouček, M. (eds.) CEEMAS 2003. LNCS (LNAI), vol. 2691, pp. 394–403. Springer, Heidelberg (2003)
35. Pavon, J., Gomez-Sanz, J., Fuentes, R.: The INGENIAS Methodology and Tools, article IX, pp. 236–276. Idea Group Publishing, USA (2005)

36. Val, E.D., Criado, N., Rebollo, M., Argente, E., Julian, V.: Service-oriented framework for Virtual Organizations. In: Proc. International Conference on Artificial Intelligence (ICAI), vol. 1, pp. 108–114. CSREA Press (2009)
37. Vazquez-Salceda, J., Dignum, V., Dignum, F.: Organizing multiagent systems. Technical Report UU-CS-2004-015, Utrecht University (2004)
38. Wagner, J., Hollenbeck, J.: Organizational Behavior. Thomson (2001)
39. Zambonelli, F., Jennings, N., Wooldridge, M.: Developing Multiagent Systems: The Gaia Methodology. ACM Transactions on Software Engineering and Methodology 12, 317–370 (2003)

Part II

Development Techniques

Model Transformations for Improving Multi-agent System Development in INGENIAS

Iván García-Magariño, Jorge J. Gómez-Sanz, and Rubén Fuentes-Fernández

Dept. Software Engineering and Artificial Intelligence
Facultad de Informática
Universidad Complutense de Madrid, Spain
ivan_gmg@fdi.ucm.es, jjgomez@sip.ucm.es, ruben@fdi.ucm.es

Abstract. Agent-Oriented Software Engineering is currently deeply influenced by the techniques and concepts of Model-Driven Development. In this context, the use of automated transformations to support software processes is not explored enough. Models are supposed to be created following the activities of a process, which standard transformations could partially perform. The reduced use of transformations for this purpose is largely due to the high effort required for their development. To overcome this limitation, this paper presents an algorithm to generate model transformations by-example. It overcomes two common limitations of these approaches: the generation of many-to-many transformation between arbitrary graphs of elements; dealing with transformation languages that do not directly support graphs of elements in their source or target models. The algorithm has been implemented for the ATLAS transformation language in the *MTGenerator* tool. The paper also illustrates how to use this tool to facilitate the support for a software process, in this case for the INGENIAS methodology.

Keywords: Model-Driven Development, Model Transformation, Model Transformation By-Example, Multi-agent System, INGENIAS.

1 Introduction

Model-driven Development (MDD) [11] has a relevant influence on *Agent-oriented Software Engineering* (AOSE). Meta-modeling techniques have already helped formalizing Multi-Agent System (MAS) specifications, and automated transformations have been proposed for their development. Nevertheless, AOSE can still take further advantage of MDD practices. One of these potential lines of research is the use of Model Transformations (MTs) as an integrating part of AOSE software processes.

The application of MTs requires to use meta-models (a model is the instance of a meta-model) specified with a meta-modeling language supported by some of the existing Model Transformation Languages (MTLs) [10]. Most AOSE methodologies are not ready for this setting. For instance, between the ten methodologies listed in [8] only three have meta-model based tools and consider some kind of transformations: PASSI, which uses UML profiles for defining

M.-P. Gleizes and J.J. Gomez-Sanz (Eds.): AOSE 2009, LNCS 6038, pp. 51–65, 2011.

its meta-model; INGENIAS and ADELFE, which are based on the ECore language [5]. The other methodologies do not strictly follow a MDD approach and use meta-models, for instance, for documentation and notation. The adoption of MDD in these methodologies should go beyond declaring the corresponding meta-models and include the modification of the existing development processes, use of meta-models by means of modeling tools and generation of artifacts with MTs. Despite the required conversion effort, there are important benefits on incorporating transformations to a process. Some of them are:

- *Knowledge capture*. A transformation is created according to the understanding of the original and transformed models. Hence, the constraints referenced by the transformation are strongly related to the intended semantics associated to the model and the development process. For instance, a transformation can be used for transitions between the different stages of the development. This way, the transformation could become the actual specification of how this transition can happen.
- *Reusability*. Transformations used for a development can be reused into another project and across different domains.
- *Decoupling*. Transformations exist independently of the tools, and act on the produced models. Hence, they can provide additional functionalities to support a methodology without changing its tools.

Following this line of research, there are already some initial results in the application of MTs to support software processes, for instance in the Tropos [4] and ADELFE [14] methodologies. However, these works do not include tool support for facilitating the definition of MTs, which makes for designers difficult to extend its use.

The experience in AOSE with MTs make clear that producing them is not trivial. It requires learning the syntax, and determining how to express left and right-hand sides of the transformation rules in terms of the input and output meta-models. Moreover, there is little support for their development, for instance debugging is usually a completely manual task. Looking for a more efficient way to produce transformations, there is research on *Model Transformation By-Example* (MTBE) [15]. This technique allows the automated generation of MTs using pairs of models that are examples of the expected inputs and outputs of those MTs. MTBE significantly reduces the cost of producing transformations. However, existent tools of MTBE [16,17] do not consider the generation of many-to-many transformation rules. These rules transform a network of concepts into another, whereas most MTBE approaches just transform one element into many. From our experience [6], the many-to-many rules are necessary in the application of MDD principles to AOSE.

This work presents several results that address the previous limitations in the application of MTs in software processes. First, it describes a MTBE algorithm able to generate MTs that contain many-to-many rules. It shows its implementation in the *MTGenerator*[1] tool for the *ATLAS Transformation Language* (ATL)

[1] Available from the *Grasia* web `http://grasia.fdi.ucm.es` (in "Software"→"MTGenerator").

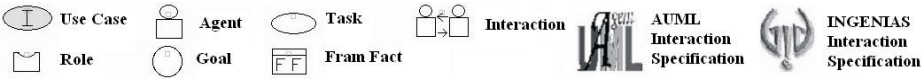


Fig. 1. Relevant elements of the *INGENIAS* notation

[2]. Second, it uses this algorithm and tool to illustrate how MTs can be used as an integrating part of software processes. The presentation considers the INGENIAS methodology and its processes [12]. Figure 1 shows some relevant elements of the *INGENIAS* notation that are used later. The paper applies MTs to generate the interactions related with use cases and to define agent skills. More transformations have been applied in our previous work [6]. These MTs constitute examples of knowledge captured within INGENIAS. With them, the way to make models evolve is made explicit. Any designer can take advantage of these MTs in any development and save effort in the definition of the specifications.

The rest of the paper is organized as follows. The next section presents the tool and its underlying MTBE algorithm. Section 3 describes in detail the two proposed examples of MTs to support the INGENIAS software process. Section 4 includes the related work about transformations and AOSE. Finally, Section 5 discusses some conclusions about the work.

2 MTBE in INGENIAS and the MTGenerator Tool

Most of AOSE designers are not used to MTLs, which makes difficult for them approaching a complete model-driven project. However, they are used to agent-oriented modeling tools. In a MTBE approach, these tools can produce example pairs of source and target models to automatically generate MTs that implement the transformation between them. Hence, MTBE makes the perfect choice for a preliminary contact with this kind of development.

This paper exemplifies the application of MTBE with *INGENIAS* modeling. As explained in the introduction, our research has found that current MTBE approaches have certain limitations that should be overcome for AOSE. Table 1

Table 1. Comparison with other MTBE approaches

Features of MTBE	Varro and Balogh [16]	Wimmer et al [17]	Our algorithm and tool
Mapping of attributes	yes	yes	yes
Propagation of links	yes	yes	yes
Negative examples	yes	no	no
Generation of constraints	no	yes	yes
Explicit allocation of target elements	yes	no	yes
Context analysis	yes	no	yes
Limit number of input elements in rules	1	1	no-limit
Limit number of output elements in rules	1	1	no-limit

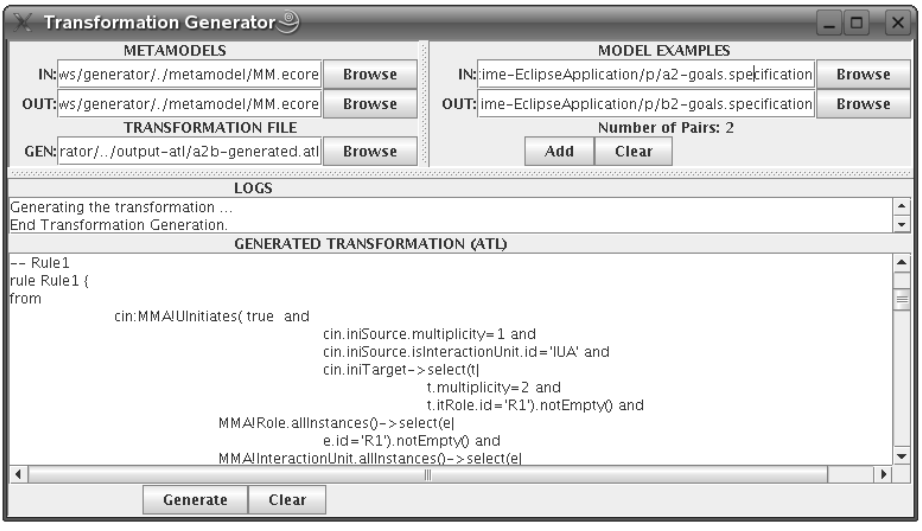


Fig. 2. *MTGenerator*, the model-transformation generator tool

compares the most relevant features of MTBE in our algorithm and other existing techniques and tools.

The prototype tool that implements this algorithm is called the *MTGenerator*. It generates ATL MTs from *INGENIAS* models. Its interface can be seen in Figure 2. The tool provides a *Graphical User interface* (GUI) where the user can select the elements required to generate the MT. First, the user must indicate the input and output meta-models of the MT by selecting their corresponding location paths in the top-left area of the GUI. These meta-models must be defined with the ECore language [5] used by ATL [2]. Then, the user can add the location paths of the prototype pairs of models in the top-right area of the tool. After the automatic generation, the tool shows some logs in the *Logs* text area, confirming that the generation has finished successfully. The bottom text area shows the generated ATL MT for examination.

Even if the user wants to improve manually the generated transformation, the tool saves time because it already provides an initial transformation as a basis for the final one. The next sub-section briefly describes the underlying algorithm of the *MTGenerator* tool.

2.1 The MTBE Algorithm

The presented algorithm produces MTs that contain rules transforming a group of elements into another group of elements. It can be implemented in most of available MTLs, as it uses common language constructions, i.e. rules from elements to graphs and constraints. Though some languages support multiple input patterns, these languages are barely used in the community. The presentation of

the algorithm uses the notation of the work in [7], which also further describes some aspects of the algorithm.

A general MT in our algorithm has the following form:

module...
rule $<$ *name* $>$ {
from $<$ *input_element* $>$ ($<$ *constraints* $>$)
to $<$ *output_elements* $>$
do {actions} }
$<$ *other_rules* $>$

The MTBE algorithm to generate such MTs corresponds to the function:

function mtbe(mm_i, mm_o : **meta-model**; pairs: $\mathcal{P}^{M_{mm_i} \times M_{mm_o}}$):MT

The algorithm takes as input two meta-models, mm_i and mm_o, and pairs of instances of those meta-models. These pairs belong to $\mathcal{P}^{M_{mm_i} \times M_{mm_o}}$, which stands for the set of sets whose elements are pairs of models instantiating the input meta-models (i.e. $\{< m_i, m_o > | m_i$ is instance of mm_i, m_o is instance of $mm_o\}$). The elements of a model m are denoted as $m.E$. The set of root elements of a model m, which are the enter points to the different graphs of elements in m, are denoted as $m.O$ and its elements as $m.O.e$. The information transfer from the input models to the output models is achieved by means of a dictionary. A **dictionary** is a set of tuples $< e_x, c_x >$, where $e_x \in E$ is an element and c_x a constraint expression of the MTL. The output of the algorithm is a MT, whose words are concatenated with the $\oplus$ operator.

1: **begin**
2: *dictionary* $= \varnothing$
3: *vars* $=$ GenerateVariables(mm_o)
4: *rules* $= \varnothing$
5: **for each** $< m_i, m_o > \in$ *pairs* **do**
6: *main* $=$ SelectMainElement(m_i.E)
7: UpdateDictionary(*dictionary*, *main*, **BasicRootConstraint**)
8: *inCons* $=$ GenerateInputConstraints(*main*)
9: . . . /* Continues later */ . . . ;

The algorithm starts with an empty dictionary, a list of variables, and a preliminary root element of the model. If there are several possible roots, then anyone is chosen. This root element is traversed with the *GenerateInputConstraints* method until reaching all connected elements by means of reference, aggregation or attribute relationships. This information is added as constraints to *inCons* which represents the input constraints of the current rule.

4: . . .
5: **for each** $< m_i, m_o > \in$ *pairs* **do**
. . . /* Continuing main loop */ . . . ;

```
 9:   for each potentialMain ∈ m_i.E such that potentialMain is a root
10:     potentialMainConstraint = CreateConstraint(potentialMain)
11:     UpdateDictionary(dictionary, potentialMain, potentialMainConstraint)
12:     inCons = inCons and GenerateInputConstraints(potentialMain)
13:   end for
14:   .../* Continues later */ ...;
```

After a reference point is chosen, which is denoted as the *main* element, the algorithm studies other possible roots and includes them as constraints in the rule. These new elements are added to the dictionary and to the input constraints. So far, all input elements have been taken into account in *inCons*, and it is the turn of considering output elements.

```
 4:          ...
 5: for each < m_i, m_o >∈ pairs do
       .../* Continuing main loop */ ...;
14:    aggregatedToRootElements = ∅
15:    outElems = ∅
16:    for each e ∈ m_o.O.e do
17:        outElems = outElems ⊕ GenerateOutputElement(dictionary, e)
           aggregatedToRootElements = aggregatedToRootElements ∪ {e}
18:    end for
19:    actions = GenerateActions (aggregatedToRootElements)
20:    rule = rule main( ⊕ inCons ⊕ ) ⊕ to ⊕ outElems ⊕
21:           imperative ⊕ actions ⊕ endrule
22:    rules = rules ⊕ rule
23:end for
```

This part of the algorithm visits all elements which are roots in the output model (i.e., the elements of $m_o.O.e$). For each of these elements, a set of MTL elements is generated. There is a set because, from the selected root, all connected elements are visited. This set contains imperative actions intended to fill in values in the attributes of generated elements. Rules are chained one to the other with ⊕ to form the final MT.

```
24: rootRule = GenerateRootRule()
25:rules = rules ⊕ rootRule
26:mt = header ⊕ vars ⊕
27:       begin ⊕ rules⊕ end
28:return mt
29:end
```

The final transformation includes the variables initially extracted and the rules obtained for each pair of models. It follows the syntax outlined at the beginning of this section.

3 Transformations in the INGENIAS Development Process

This section considers the application of the *MTGenerator* tool, which implements the previous MTBE algorithm, to generate MTs that support an AOSE software process. Specifically, this paper considers the different ways of instantiating the INGENIAS [12] meta-model to generate MAS specifications and their artifacts. Here, an artifact stands for a diagram or part of it containing a piece of information relevant for the development.

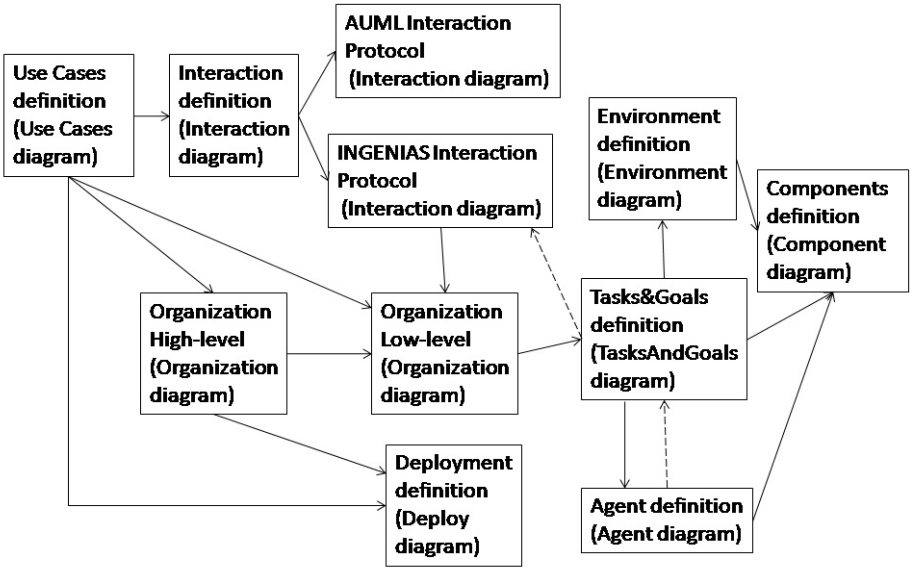


Fig. 3. INGENIAS development processes. The arrows indicate the order of artifact generation, though inverse navigation is also possible. The dotted arrows indicate the influence of a diagram into another.

As a result of the experience in the development of MAS with INGENIAS, this work has identified a recurrent order in the generation of the different artifacts. This dependency graph is shown in Figure 3. In spite of the variety of options, the INGENIAS modeling usually starts with use cases. The use cases can be refined either with interactions or organizations. However, both possibilities are focused on specifying the organization structure. After them, those organizations are refined with the tasks and goals of their agents. This refinement implies declaring the inputs and outputs of the tasks and their relationships with the goals. The next step is the simultaneous specification of agents and elements in the environment. The agents are linked with roles, which are responsible of the execution of some tasks. The environment contains the internal and external applications,

which produce events that trigger the execution of certain tasks. Finally, specifications describe some low-level details of the implementation of both tasks and applications. INGENIAS makes it with components that associate pieces of programming code to elements in specifications.

Each of the previous steps between two types of information, i.e. diagrams, can be assisted with several MTs. For the sake of brevity, the next sub-sections only present three of these MTs. The reader can find the application of more MTs generated with the *MTGenerator* tool in our previous work [6].

3.1 Transformations for Realizing Use Case Behaviors

The realization of use cases in INGENIAS can follow several alternatives. One of them is to expand the definition of the interactions that realize it. For each interaction, a protocol has to be defined. In the case of this MT, the specification of the protocol uses a custom notation of INGENIAS which permits combining message passing primitives with task execution. Using model transformations and starting from the initial use case, it is possible to define prototypes for the resulting concepts and the relationships between them and the original concepts.

The first MT, called *UseCase2Interaction*, creates the definition of an interaction from a use case. Figure 4 shows its model example prototypes. The MT defines an interaction that communicates the agents and roles related with a particular use case. Annex A shows the code of the generated ATL transformation.

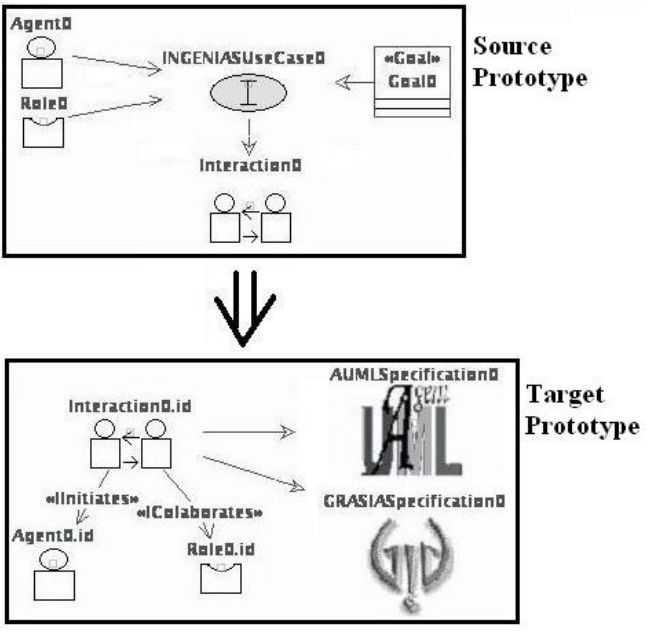


Fig. 4. Model Examples for the *UseCase2Interaction* model transformation

In the code of Annex A, one can observe that *Rule 1* expresses the source prototype with constraints. For instance, the relationship between a use case and its goal is represented with a constraint that refers to the *UseCasePursues* relationship type. Moreover, each element of the target model prototype is generated as an output element of the rule, like the *AUMLSpecification* element. Note that, according to the INGENIAS metamodel, the relationships are represented with several elements (for the relationship bodies and their ends). For instance, the *UseCasePursues* relationship type has as ends *UseCasePursuessource* and *UseCasePursuestarget*. The MTBE algorithm takes these elements into account even if the editor of the *INGENIAS Development Kit* (IDK) tool does not graphically present them, because they are necessary to define correctly the MT.

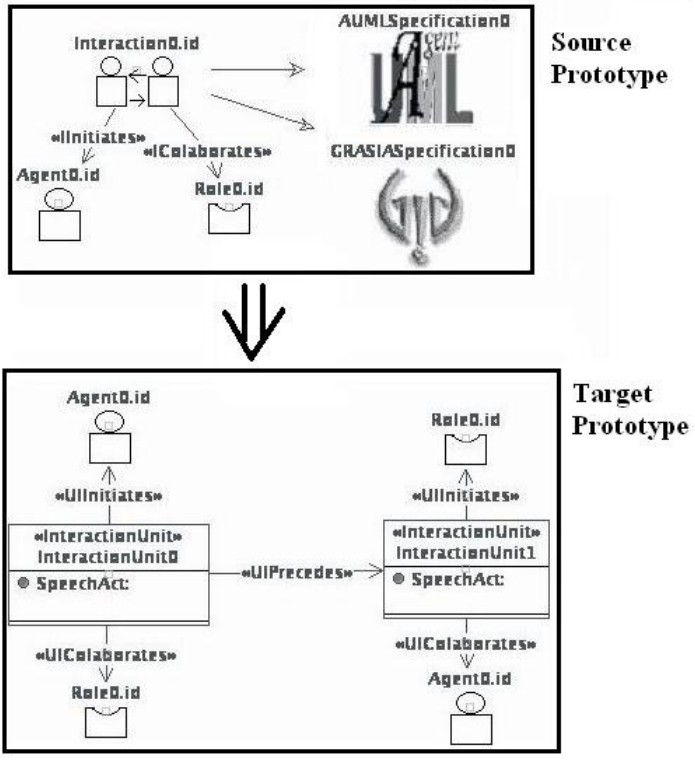


Fig. 5. Model Examples for the *InteractionDefinition2InteractionProtocol* model transformation

Another MT, called *InteractionDefinition2InteractionProtocol*, creates a basic protocol from an interaction definition. The pairs of models appear in Figure 5. The resulting protocol includes two interaction units, in which there are a request and a response. This protocol is only a starting point, and designers must use it to complete the specification.

3.2 Transformation for Skill Definition

Defining the skills of an agent is a basic activity in most AOSE methodologies. A skill can be regarded as the capability to fulfill a goal. In INGENIAS, this implies: defining a task; declaring that the agent plays a role responsible of the task execution; and associating the goal to the task. The task can use resources or additional applications. INGENIAS applications are wrappers that allow agents to access non-agent software. As a result of task execution, new information is produced and asserted into the agent mental state.

Using again MTBE, this knowledge can be synthesized into a pair of prototypes and incorporated into a MT. Figure 6 presents these examples. They correspond to a source agent that acquires a new skill. The skill is represented in the target model as a goal pursued by the agent and whose satisfaction depends on a task. The relationship between agent and task is indirect through a new agent's role that is responsible of the task. The generated MT is applied to all the agents by default. Designers can apply this MT to only one agent if necessary, by adding a simple additional constraint with the agent identifer (i.e. $cin.id =< AgentID >$).

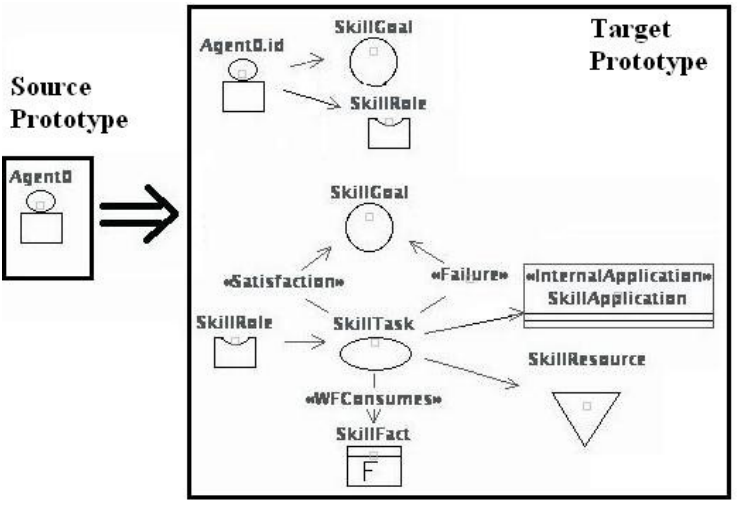


Fig. 6. Model Examples for the *AddSkill* model transformation

3.3 Evaluation of the Use of Transformations on an AOSE Methodology

The experience with INGENIAS allows us to extrapolate and highlight the following facts about the application of MTs in AOSE methodologies:

– *Traceability problem.* Some inconsistences may occur when modifying models which were generated from MTs. This problem will be studied in the future. MDD usually recommends in this cases modifying the affected MTs.

– *Reducing costs.* MTs can be applied several times in a MAS specification, producing many concepts and saving effort to designers. However, the reduction of costs must still be quantitatively measured.
– *Reusability across domains.* The presented MTs are reusable, but this does not always happen. Examples of potential non-reusable MTs would be those requiring explicit values in input pairs.

4 Related Work

There are already some examples of the application of MTs to support AOSE software processes. This section discusses some of the most relevant.

The Tropos [4] methodology applies MTs for different tasks. Tropos conceives development as an incremental process that adds new elements refining existing actors, goals and plans. This refinement is organized in five major phases or disciplines: *early requirements, late requirements, architectural design, detailed design* and *implementation.* Bresciani et al. [4] apply MTs to perform standard refinements of requirements in the early requirements phase. Perini and Susi [13] use MTs during design, in what they called *synthesis.* The synthesis transforms a model into another of a lower level of abstraction, specifically, it creates elements in the detailed design model from the architectural design model. One of the main goals of that work is to keep synchronization when applying MTs. Moreover, since the detailed design model is expressed with UML activity and sequence diagrams, that work aims at exploiting the UML 2.0 meta-model and to maintain the traceability between models with it. The mentioned Tropos-to-UML MT specifies several one-to-many rules: they transform a plan into an action, several nodes and several control flows. The group of target elements can vary due to certain constraints. These Tropos MTs are expressed with the DSTC language proposal for MOF QVT, and these MTs are executed with a MT engine called Tefkat[2].

Moreover, Amor, Fuentes and Vallecillo [1] apply the Model-Driven Architecture (MDA) [11] principles to the Tropos methodology. In particular, their MTs receive input from the Platform-Independent Models (PIM), expressed with the Tropos meta-model, and create a Platform Specific Model (PSM), expressed with the Malaca MAS language.

In the same line of research, the ADELFE [3] methodology uses a MT to refine PIM into more specific models. In particular, its AMAS-ML is the abstract language whereas the μADL (micro-Architecture Description Language) [14] is a platform-specific language that expresses operating mechanisms of agents (similar to non-functional concerns). These mechanisms are related with the concepts involved in the creation and maintenance of the agent knowledge, such as perceptions and actions. In this manner, the behavior of cooperative agents, defined in the *AMAS-ML* model, is separated from the operating concern.

[2] Tefkat is part of the Pegamento project of the DSTC in the University of Queensland `http://www.dstc.edu.au/Research/Projects/Pegamento/tefkat/index.html`

The work in [9] also proposes a language for MAS with a high-level abstraction language. Models of this language could be successively transformed into others more specific until code is generated.

All the aforementioned works use MTs to support software processes, but they typed these without assistance of any tool. In contrast, our approach, which pursues the same support goal, uses a tool for assisting MAS designers in creating MTs and automatically generates them.

5 Conclusions

The integration of transformations to support AOSE methodologies still raises relevant challenges: high development efforts, low level of abstraction when compared with models, and need of adapting methodologies to the principles of MDD.

INGENIAS has introduced some advances in this line of research for AOSE. It is an example of methodology that bases its tools in meta-models, and has reoriented its process to integrate MTs. However, this has required an important reconversion effort from previous versions of the methodology. For this migration, the availability of a tool to generate MTs by example has been key, since it has increased the productivity of engineers. The MTBE algorithm of the proposed tool overcomes some common limitations of these approaches, specifically that they cannot generate transformations between arbitrary graphs of elements. The resulting tool has facilitated the integration of MTs as a relevant and standard support of the development process. The example transformations introduced in this paper, as well as the tools that made possible to achieve these results, are available from the *Grasia* web.

Other methodologies have chosen to create from scratch new languages or getting rid of the old tools. However, this work considers that this implies an important lost of previous work that should be avoided when possible.

Future research of this work includes two lines. The main drawback of the algorithm is that it cannot specify negative examples, that is, elements that should not appear in the source model of the prototype pair. This problem has been already considered in the literature and solutions will be added in extensions of the algorithm. About methodologies, there is still lack of experience in the development of processes that consider as an integrating part of their resources MTs. More work can lead to the definition of libraries of MTs reusable in AOSE between different methodologies and processes.

Acknowledgements. This work has been done in the context of the project "Agent-based Modelling and Simulation of Complex Social Systems (SiCoSSys)", supported by Spanish Council for Science and Innovation, with grant TIN2008-06464-C03-01. Also, we acknowledge support from the "Programa de Creación y Consolidación de Grupos de Investigación UCM-BSCH" (GR58-08).

References

1. Amor, M., Fuentes, L., Vallecillo, A.: Bridging the Gap Between Agent-Oriented Design and Implementation Using MDA. In: Odell, J.J., Giorgini, P., Müller, J.P. (eds.) AOSE 2004. LNCS, vol. 3382, pp. 93–108. Springer, Heidelberg (2005)
2. ATL Transformations Repository (provided by the Eclipse Web), http://www.eclipse.org/m2m/atl/atlTransformations/ (available on September 30, 2009)
3. Bernon, C., Camps, V., Gleizes, M.P., Picard, G.: Engineering Adaptive Multi-Agent Systems: The ADELFE Methodology. In: Henderson-Sellers, B., Giorgini, P. (eds.) Agent-Oriented Methodologies, pp. 172–202. Idea Group Publishing, NY (2005)
4. Bresciani, P., Perini, A., Giorgini, P., Giunchiglia, F., Mylopoulos, J.: Modeling Early Requirements in Tropos: A Transformation Based Approach. In: Wooldridge, M.J., Weiß, G., Ciancarini, P. (eds.) AOSE 2001. LNCS, vol. 2222, pp. 151–168. Springer, Heidelberg (2002)
5. Moore, B., et al.: Eclipse Development using Graphical Editing Framework and the Eclipse Modelling Framework. IBM Redbooks (2004)
6. García-Magariño, I., Gómez-Sanz, J.J., Fuentes-Fernández, R.: INGENIAS Development Assisted with Model Transformation By-Example: A Practical Case. In: 7th International Conference on Practical Applications of Agents and Multi-Agent Systems (PAAMS 2009). Advances in Soft Computing, vol. 55, pp. 40–49. Springer, Heidelberg (2009)
7. García-Magariño, I., Gómez-Sanz, J.J., Fuentes-Fernández, R.: Model transformation by-example: an algorithm for generating many-to-many transformation rules in several model transformation languages. In: Paige, R.F. (ed.) ICMT 2009. LNCS, vol. 5563, pp. 52–66. Springer, Heidelberg (2009)
8. Henderson-Sellers, B., Giorgini, P.: Agent-Oriented Methodologies. Idea Publishing, USA (2005)
9. Jung, Y., Lee, J., Kim, M.: Multi-agent based community computing system development with the model driven architecture. In: AAMAS 2006: Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems, pp. 1329–1331. ACM, New York (2006)
10. Mens, T., Van Gorp, P.: A taxonomy of model transformation. Electronic Notes in Theoretical Computer Science 152, 125–142 (2006)
11. Object Management Group: Model driven architecture guide version 1.0.1, omg document ad/2002-04-10 (2002) (June 2003)
12. Pavón, J., Gómez-Sanz, J.J., Fuentes, R.: The INGENIAS Methodology and Tools. In: Henderson-Sellers, B., Giorgini, P. (eds.) Agent-Oriented Methodologies, pp. 236–276. Idea Group Publishing, NY (2005)
13. Perini, A., Susi, A.: Automating Model Transformations in Agent-Oriented Modelling. In: Müller, J.P., Zambonelli, F. (eds.) AOSE 2005. LNCS, vol. 3950, pp. 167–178. Springer, Heidelberg (2006)
14. Rougemaille, S., Migeon, F., Maurel, C., Gleizes, M.-P.: Model Driven Engineering for Designing Adaptive Multi-Agents Systems. In: Artikis, A., O'Hare, G.M.P., Stathis, K., Vouros, G.A. (eds.) ESAW 2007. LNCS (LNAI), vol. 4995, pp. 318–332. Springer, Heidelberg (2008), http://www.springerlink.com
15. Varró, D.: Model transformation by example. In: Wang, J., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 410–424. Springer, Heidelberg (2006)

16. Varró, D., Balogh, Z.: Automating model transformation by example using inductive logic programming. In: Proceedings of the 2007 ACM Symposium on Applied computing, pp. 978–984 (2007)
17. Wimmer, M., Strommer, M., Kargl, H., Kramler, G.: Towards Model Transformation By-Example. In: Proceedings of the 40th Annual Hawaii International Conference on System Sciences, vol. 40(10), p. 4770 (2007)

A Generated ATL Transformation for the Usecase2Interaction Transformation

```
module a2b_2; -- Module Template
create OUT : MMB from IN : MMA;
-- For the two new built entities and relations
helper def: newEntities: Sequence(MMB!GeneralEntity)=Sequence{};
helper def: newRelations: Sequence(MMB!GeneralRelation)=Sequence{};

---------------------------- Rules for Patterns ----------------------

-- Rule1
rule Rule1 {
from
    cin:MMA!UMLDescribesUseCase(
            cin._view_type='INGENIAS' and
            cin.UMLDescribesUseCasesource._view_type='INGENIAS' and
            cin.UMLDescribesUseCasesource.entity._view_type='INGENIAS' and
            cin.UMLDescribesUseCasesource.attributeToShow='0' and
            cin.UMLDescribesUseCasetarget->select(t|
                t._view_type='INGENIAS' and
                t.attributeToShow='0' and
                t.UMLDescribesUseCasetargetInteraction._view_type='INGENIAS')
                    .notEmpty() and MMA!UseCasePursues.allInstances()->select(e|
            e._view_type='INGENIAS' and
            e.UseCasePursuessource._view_type='INGENIAS' and
            e.UseCasePursuessource.attributeToShow='0' and
            e.UseCasePursuessource.UseCasePursuessourceGoal._view_type='UML' and
            e.UseCasePursuestarget->select(t|
                t._view_type='INGENIAS' and
                t.entity._view_type='INGENIAS' and
                t.attributeToShow='0').notEmpty()).notEmpty() and
        MMA!Goal.allInstances()->select(e|
            e._view_type='UML').notEmpty() and
        MMA!Interaction.allInstances()->select(e|
            e._view_type='INGENIAS').notEmpty() and
        MMA!INGENIASUseCase.allInstances()->select(e|
            e._view_type='INGENIAS').notEmpty() )
to
    Interaction0xid:MMB!Interaction(
        _view_type<-INGENIAS,
        id<-cin.UMLDescribesUseCasetarget->select(t|t._view_type='INGENIAS' and
t.attributeToShow='0').first().UMLDescribesUseCasetargetInteraction.id,
        TransferredInfo<-),
    AUMLSpecification0:MMB!AUMLSpecification(
        _view_type<-INGENIAS,
        id<-AUMLSpecification0),
    GRASIASpecification0:MMB!GRASIASpecification(
        _view_type<-INGENIAS,
        id<-GRASIASpecification0),
    newid4:MMB!IHasSpecsource(
        _view_type<-INGENIAS,
        id<-newid4,
        attributeToShow<-0,
        IHasSpecsourceInteraction<-Interaction0xid),
    newid5:MMB!IHasSpectarget(
```

```
        _view_type<-INGENIAS,
        id<-newid5,
        entity<-AUMLSpecification0,
        attributeToShow<-0),
    9:MMB!IHasSpec(
        _view_type<-INGENIAS,
        id<-9,
        end<-'',
        IHasSpecsource<-newid4,
        IHasSpectarget<-newid5),
    newid6:MMB!IHasSpecsource(
        _view_type<-INGENIAS,
        id<-newid6,
        attributeToShow<-0,
        IHasSpecsourceInteraction<-Interaction0xid),
    newid7:MMB!IHasSpectarget(
        _view_type<-INGENIAS,
        id<-newid7,
        entity<-GRASIASpecification0,
        attributeToShow<-0),
    11:MMB!IHasSpec(
        _view_type<-INGENIAS,
        id<-11,
        end<-'',
        IHasSpecsource<-newid6,
        IHasSpectarget<-newid7)
do{
         thisModule.newEntities<-thisModule.newEntities.append(Interaction0xid);
         thisModule.newEntities<-thisModule.newEntities.append(AUMLSpecification0);
         thisModule.newEntities<-thisModule.newEntities.append(GRASIASpecification0);
         thisModule.newRelations<-thisModule.newRelations.append(9);
         thisModule.newRelations<-thisModule.newRelations.append(11);
    }
}

------------- Rule for Specification -----------------------------

rule Specification{
    from
        cin:MMA!Specification
    to
        cout:MMB!Specification
    do{
        -- We add the new entities and the new relations
        for(e in thisModule.newEntities){
            cout.entities<-e;
        }
        for(r in thisModule.newRelations){
            cout.relations<-r;
        }

    }
}
```

Automated Testing for Intelligent Agent Systems

Zhiyong Zhang, John Thangarajah, and Lin Padgham

RMIT University, Melbourne, Australia
pdt@cs.rmit.edu.au

Abstract. This paper describes an approach to unit testing of plan based agent systems, with a focus on automated generation and execution of test cases. Design artefacts, supplemented with some additional data, provide the basis for specification of a comprehensive suite of test cases. Correctness of execution is evaluated against a design model, and a comprehensive report of errors and warnings is provided to the user. Given that it is impossible to design test suites which execute all possible traces of an agent program, it is extremely important to thoroughly test all units in as wide a variety of situations as possible to ensure acceptable behaviour. We provide details of the information required in design models or related data to enable the automated generation and execution of test cases. We also briefly describe the implemented tool which realises this approach.

1 Introduction

The use of agent technology for building complex systems is increasing, and there are compelling reasons to use this technology. Benfield [1] showed a productivity gain of over 300% using a BDI (Belief Desire Intention) agent approach, while Padgham and Winikoff calculated that a very modest plan and goal structure provides well over a million ways to achieve a given goal [2, p.16], providing enormous flexibility in a modular manner. However the complexity of the systems that can be built using this technology, does create concerns about how to verify and validate their correctness. In this paper we describe an approach and tool to assist in comprehensive automated unit testing within a BDI agent system. While this approach can never *guarantee* program correctness, comprehensive testing certainly *increases confidence* that there are no major problems. The fact that we automate both test case generation, as well as execution, greatly increases the likelihood that the testing will be done in a comprehensive manner.

Given the enormous number of possible executions of even a single goal, it is virtually impossible to attempt to test all program traces. Once interleaved goals within an agent, or interactions between agents are considered, comprehensive testing of all executing becomes clearly impossible. Instead, we focus on testing of the basic units of the agent program - the beliefs, plans and events (or messages). Our approach is to ascertain that no matter what the input variables to an entity, or the environment conditions which the entity may rely on, the entity behaves "as expected". Our notion of expectations is obtained from design artefacts, produced as part of an agent design methodology. We focus in this paper on the details of how we determine variable values for test cases to provide comprehensive coverage, and the representations and mechanisms we use to allow us to automate the process (though also providing mechanisms for user specified test cases where desired.) We build on our previous work in [3] which described a

M.-P. Gleizes and J.J. Gomez-Sanz (Eds.): AOSE 2009, LNCS 6038, pp. 66–79, 2011.

basic architecture and approach. However we address in this paper some of the details necessary to effectively realize that approach.

There are two specific aspects of automated test generation and execution which we focus on in this paper. The first is how to specify values relevant to a particular unit to be tested, and how to generate appropriate test cases that adequately cover the space of value combinations for all relevant variables. Doing this appropriately is critical in order to be able to have confidence that the testing process was thorough. The second important aspect is the setting up of the environment such that the unit to be tested can be executed appropriately. One aspect of this is to ensure that units which are depended on are tested first, and then included in the environment for the unit being tested. However in many real applications there may be interaction with an external program within a plan unit (e.g. a query to an external database of a web service). There may also be interaction with another agent within a testing unit. In these cases we must ensure that access to the external program is set up prior to testing, and that there is some mechanism for dealing with necessary interaction with other agents. We note that most interaction with other agents would not be *within* a testing unit. Typically an agent would generate a message in one plan, and handle it another, in which case there is no need to model the other agent in any way.

In the following sections we first provide an overview of the testing process of our previous work [3], the framework on which this work is based. In section 3, we describe the automated process for test case generation. Section 4 then covers the details needed to provide the environment where the units can be executed, including how to automatically generate *mock agents* to simulate interaction if necessary. We provide in section 5 a brief discussion of the implemented system and some results from its use, concluding in section 6 with a brief comparison to other work and directions for ongoing development.

2 Testing Process Overview

In [3], we present a model based framework for unit testing in agent systems, where the models used are provided by the design diagrams of the Prometheus methodology, a well established agent development methodology [2]. The Prometheus methodology, and associated Prometheus Design Tool (PDT) produces graphical models capturing relationships between basic entities such as goals, plans and beliefs. It also produces *descriptors* which collect together a range of design information for each design entity. The detailed design entities map closely to implementation entities and are utilized for generation of skeleton code. They can also be used for generation of test cases and analysis of testing results.

Figure 1 outlines the components of an agent that is developed using the Prometheus methodology. These components are typical for most systems that follow the Belief Desire Intention (BDI) model of agency [4]. An agent contains *plans*, *events* and *beliefs* and can have *capabilities* that encapsulate them for modularity. Percepts (external input to the system), messages (from other agents) and internal events are all considered as events in agent development tools such as JACK [5] and we use this same generalization. In brief, the agent uses its plans to handle events as they occur and may query and

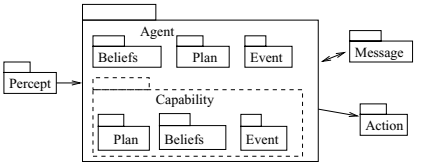


Fig. 1. Agent Component Hierarchy in Prometheus

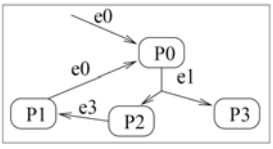


Fig. 2. Plan Dependencies

update its beliefs as needed during this process. Belief changes may automatically generate events. Given the above components, the units to be tested are the *events*, *plans* and *beliefs*. This extends the work in [3] which covered only events and plans.

Events are tested with respect to *coverage* and *overlap*. Coverage refers to the expectation that an event will always be handled by at least one plan in any situation. *Overlap* refers to the possibility that there may be more than one plan that is applicable for handling a given event, in some situation. While it is not *necessarily* an error to have an event without full coverage, or with overlap, these are common sources of error in agent programs [6]. Consequently PDT prompts developers to consider and to specify expected coverage and overlap as part of design of events. Test case results are then compared against the design specification.

A plan definition includes the type of event which triggers it, a context condition that determines its applicability in a given situation, and a plan body. The body of a plan may contain actions (considered atomic) and/or post sub-events (sub-tasks). Plans are tested for the following:

- Does the plan get triggered by the event that it is supposed to handle? If it does not, then either some other plan always handles it or there is an inconsistency between design and implementation.
- Is the context condition valid in some situations? If no context condition is specified then the plan is always applicable. However, if a context condition is specified then there must be at least some situation where it is satisfied for the plan to be applicable. If it is satisfied in any situation that could also indicate an error as it that is the equivalent of an empty context condition.
- Does the plan post the events that it should? Events are posted from a plan to initiate sub-tasks (or messages sent to other agents). If some expected events are never posted, we need to identify them as this may be an error. Here we can only check if the design matches the code, and can not check, for example, if the internal logic of the plan is correct/sensible or if the event/message contains the correct data.
- Does the plan complete? It is possible to determine whether the plan executed to completion. However, we cannot determine whether the plan completed successfully as the information required to determine the success of the plan is not currently contained in an extractable format in the design specification.

If a plan A posts a sub-task (sub-event) that is handled by plan B then the success of plan A is dependent on plan B, which should in general be tested before A. There may however be cyclic dependencies. For example, in Figure 2 $P0, P2$ and $P1$ are what is termed a *Plan Cycle* [3].

Plan cycles are a special case of testing plans, in that, in addition to testing for the aspects specific to plans, each plan cycle is considered as a single unit and tested for the existence and the termination of the cycle at run-time. If the cycle doesn't terminate after a maximum number of iterations this generates a warning to the user. If the cycle never occurs a warning is given as the cycle may have been a deliberate design decision.

While in the general case, events are generated from external input or from within a plan body, events may also be generated automatically when certain beliefs (conditions) become true. For example, if a robot agent believes the energy level to be less than 10% it may generate an event to re-charge. In agent systems like JACK [5] such events are implemented as database call-backs, other systems may implement them in the form of active databases, for example [7]. For every belief that generates such events, we test if the appropriate events get posted in the situations as specified in the design. This aspect of testing beliefs is an addition to the framework presented in [3].

The testing process consists of a number of steps which we outline briefly below.

1. Determine the order in which the units (events, plans, plan cycles and belief-sets) are to be tested. If a plan contains sub-tasks the success of the plan is partially dependent on the plan(s) that satisfy the sub-tasks. For example, in Figure 2 the success of plan $P0$ depends on the success of $P2$ or $P3$. The order of testing is therefore bottom-up where a unit is tested before the units that depend on it.
2. Develop test cases with suitable input value combinations as described in section 3.
3. Augment the source code of the system under test with special testing code to provide appropriate information to the testing system.
4. Execute the test cases, collect and analyze results and produce a detailed report. The report contains both errors and warnings, where the warnings are things which are likely to be errors, but which could be intended, or be due to lack of an appropriate test case (e.g. failure to have a context condition become true in any test, may be that a new specialized test needs to be added).

All of the above steps are automated and can be performed on a partial implementation of the system if desired. Figure 3, provides an abstract view of the testing framework for a plan unit. There are two distinct components: the *Test driver* and the *Subsystem under test*. The former contains the *Test Agent*, that generates the test cases and executes the test cases by sending an *Activation Message* to the *Test-Driver Plan*. This plan is embedded into the Subsystem under test during the code augmentation phase, and is

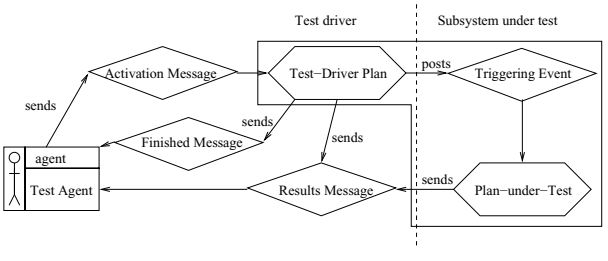


Fig. 3. Abstract Testing Framework for a Plan

responsible for setting up the test input and activating the *Subsystem under test* via a *Triggering Event*.

The subsystem contains the units to be tested and other parts of the system required to activate them. These units under test are able to send *Results Message*s back to the *Test Agent* via testing specific code inserted into them. After all testing is complete the *Test Agent* generates the test report.

3 Test Case Generation

Test cases are automatically generated for each unit, by extracting relevant variables from the design documents and generating variable value combinations. It is important to ensure that there is comprehensive coverage of the different possible situations the unit may execute in, so the choice of variable values is very important. At the same time, it is impossible to test all values, so choices must be made carefully. We also need to know which variables (in the environment, in the agent's beliefs, or in the unit itself) affect the particular unit. This is specified in the design documentation. These specifications enable automatic extraction of the variables.

Some examples are:

[event-var, book.bookName, string, !=null]
[belief-var, (StockDB, numberInStock), int, >0]

Table 1. Example of equivalence classes

name	index	domain	valid	minimal	normal	comprehensive
bookID	EC-1	$(-\infty, 0)$	no	N/A	-1	-1
	EC-2	$[0, +\infty)$	yes	0	0, 1	0, 1
stock	EC-1	$(-\infty, 0)$	no	N/A	-1	-1
	EC-2	[0, 200]	yes	0, 200	0, 1, 199, 200	0, 1, 100, 199, 200
	EC-3	$(200, +\infty)$	no	N/A	201	201
price	EC-1	$(-\infty, 0.0]$	no	0.0	-0.1, 0.0	-0.1, 0.0
	EC-2	(0.0, 90.0]	yes	90	0.1, 89.9, 90	0.1, 45.0, 89.9, 90.0
	EC-3	$(90.0, +\infty)$	no	90.1	90.1	90.1

Having automatically extracted the variables from the design for a particular unit, the next step in the testing process is to generate values for these variables which will test the unit. In many cases it is not possible to test all possible values as the domain may be infinite or very large. We use the standard notions of *Equivalence Class Partitioning* and *Boundary Value Analysis* [8, p.67] to generate a limited set of values for each relevant variable. An Equivalence Class (EC) is a range of values such that if any value in that range is processed correctly (or incorrectly) then it can be assumed that all other values in the range will be processed correctly (or incorrectly). Boundary values mark the (non-infinity) ends of an EC, and are often values which cause errors. Some approaches to testing use only valid ECs [9, p.98], while others improve robustness by also using invalid ECs [9, p.99] [10, p.39] [8, p.67] .

In our approach we allow the user to select from three different levels for choice of values for variables:

1. Minimal level is restricted to valid values, and uses boundary values for the valid ECs.
2. Normal level uses both valid and invalid ECs, selecting boundary values, and values close to the boundary, for each EC.
3. Comprehensive level also adds a nominal value in the mid-range of each EC.

To illustrate the above let us consider the following variables:

[agent-var, bookID, int, ≥ 0]
[agent-var, stock, int, ≥ 0, ≤ 200]
[agent-var, price, float, >0.0, $<= 90.0$]

Table 1 gives their ECs, and the choice of values from the three levels. Having chosen test values for each relevant variable these must be combined in various ways to produce test cases. Some approaches to testing simply ensure that for each variable, there is at least one test case with each of the values chosen [9, p.98] [8, p.71] [11, p.55]. A more thorough approach recognizes that many errors are a result of interactions between variable values, and takes the cross product of all values for each variable. The problem is that the latter quickly gives a very large number of test cases. The first approach caps the number of test cases at the largest number of values for any variable. The second is the cartesian product of the number of values for each variable, which quickly explodes. For example if we have five variables, each with five values, it gives over 3,000 test cases.

A commonly used approach to reduce the number of combinations is combinatorial design [12]. This approach generates a new set of value combinations that cover all n-wise (n>=2) interactions among the test parameters and their values in order to reduce the size of the input data set. We use the *Combinatorial Testing Service* software library of Hartman and Raskin[1] which implements this approach, in our system. However we apply this reduction only to test cases involving invalid values. We use all combinations of valid values, on the assumption that, for agent systems, it is interactions between valid variable values which will cause different aspects of the code to be activated (most commonly different plans chosen), and that covering all such interactions is necessary in order to adequately evaluate if the system is behaving correctly.

As with choice of values we allow the user to choose between different levels of thoroughness in testing. We provide the following options:

1. Basic level takes the cartesian product of valid values, and then adds one additional case for each invalid value, based on the assumption that invalid values do not require so rigorous testing as valid values.
2. Extended level supplements the results with all valid value cartesian products not in the set obtained using the pairwise reduction of combinatorial design [12].

In addition to the test cases that are auto-generated as described above, the user may wish to specify additional test cases using his domain and design knowledge. The testing framework accommodates this by means of a *Test Case Input* window in PDT for a given unit under test.

[1] `http://www.alphaworks.ibm.com/tech/cts` (obtained July 2006, technology now retired).

4 Test Case Execution

In order to execute a test case we need to first set up the environment so that it is able to run. Depending on the unit this may include such things as setting up sockets for communicating with an external process, setting up a global data structure that will be accessed, and so on. Then we also need to assign the values we have chosen for the relevant test case, to the appropriate variables, which requires knowing how these variables are implemented in the code. In this section we describe the details of the execution process, and the information representation which we require in order for this to be fully automated.

Initialization Procedures

The testing process tests each individual unit. Prior to executing the test case for that unit however, it may be necessary to perform some initialization routines such as setting up connections to external servers, populating databases, initializing global variables and so on.

In the testing framework we allow these routines to be specified as initialization procedures for each unit in the *Unit Test Descriptor* (see Figure 5). The *Unit Test Descriptor* captures testing specific properties of the unit which is in addition to the usual design descriptor that specifies the properties of the unit. We allow multiple procedures to be specified where each are in the following format:

<order, owner_object, is_static, function_call, comment>.

Figure 4, is an example of such a specification. In this example, the unit requires the $initBookDB$ function of the $StockAgent$ to be called as the first initialization procedure. The method is static, hence can be invoked directly from the $StockAgent$ class. When complete the next method $initConnAmazon$ is executed.

order	owner object	is static	function call	comment
1	Stock Agent	yes	initBookDB()	method to populate the books database
2	BuyPlan	no	initConnAmazon()	sets up connection to Amazon.com

Fig. 4. Example specification of initialization procedures

Variable Assignment

In section 3 test cases were defined by generating values for each specified variable of the unit under test. To execute the test case these values must be assigned to the variables. The technique for assigning a value to a variable may vary depending on how the variable is coded in the implementation of the system. For example, a variable that is private to an object[2] needs to be set via the object's mutator functions. Because the testing process is fully automated, it is necessary that there be some specification of an *assignment relation* to allow appropriate assignment of variable values to the implementation of the variable in the code.

[2] In terms of Object Oriented programming.

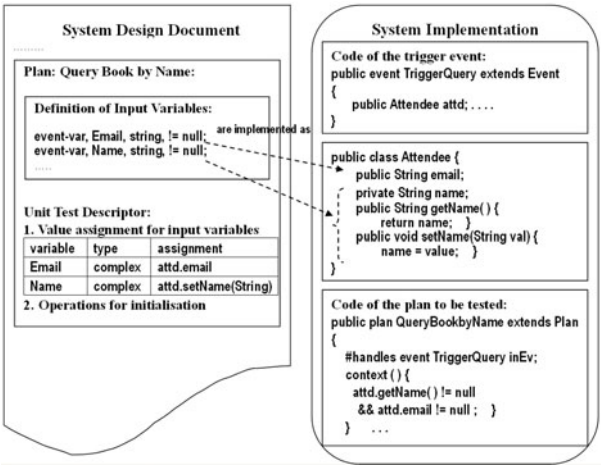


Fig. 5. Illustration of complex variables

This assignment relation is specified in the Unit Test Descriptor of the unit and takes the following form for each variable:

<variable-name, type, assignment>

where the *type* is classified as *simple, complex, belief* or *function* based on how the variable is implemented in the source code of the system under test. The *assignment* relation depends on these types and we describe them below:

A ***simple*** variable is implemented as a public variable, which could be directly set, or a private variable that is set via a public mutator function. In general, if no assignment relation is specified for a variable, the assumed default is a simple variable that is publicly accessible.

A ***complex*** variable is one that is part of a nested structure, such as an attribute of an object, which may in turn be part of another object and so on. For example, in Figure 5, the variables $Email$ and $Name$: are attributes of the $attd$ object of type $Attendee$ in the triggering event of the plan. The assignment relation for the $Email$ variable is $attd.email$ as it is a public attribute of that object and can be set directly. The $Name$: however, has to be set via the $att.setName(String)$ method.

Belief variables are fields of a particular belief-set. For example:

[belief-var, (StockDB, bookID), int, $==1$]
[belief-var, (StockDB, inStock), int, >0 , <5]

They do not require an assignment function as the technique for assigning variables would be the same for any field of the belief. That is, create and insert a record with the values generated for the belief variables in concern and random values for the rest of the fields for that record. For example, in Figure 6 the first test case is setup by creating and inserting a record for the $StockDB$ beliefset with fields $bookID$ and $inStock$ set to 1

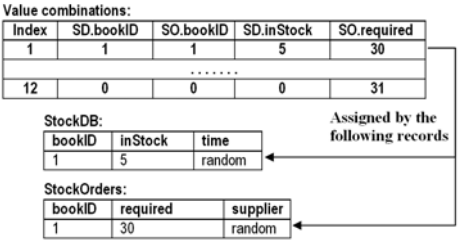


Fig. 6. Illustration of belief variables

and 5 respectively, and creating and inserting a record for the $StockOrders$ beliefset with fields $bookID$ and $required$ set to set to 1 and 30 respectively[3].

Although the automated test cases contain randomly generated values for the belief fields that are not specified as unit test variables, as with all the test units the user may specify additional test cases (value combinations) prior to executing the tests.

A special case of belief variables is when historical information is required. For example, if the context condition of the plan contains the following:

StockDB.getStockAt(t_1) $>$ *StockDB.getStockAt(t_2)*
the design document will contain:
[belief-var, StockDB, numberInStock, int, >0].

If we follow the approach above, value combinations will be generated for $numberInStock$ but only one record will be inserted per test case, which would be insufficient to evaluate that particular context. In the general case, this situation is where the belief-set is to be populated with multiple rows. There are two alternatives for testing such situations in our testing framework.

The first is for the user to specify a test case manually for this belief. When specifying a test case manually for a belief, the user is given the option to add as many rows as desired. This approach however could be tedious if many rows are to be inserted. The second approach is to provide a method for populating the database as an initialization procedure of the unit under test.

There may be instances where a variable in the design is realized by a function in the implementation. We term these variables as ***function*** variables. For example, the variable *total_order_cost* which requires some calculation. It is not possible to set the value of these variables as it depends on the value returned by the function. This value may depend on a number of other variables, some local to the function others outside. The current testing framework ignores such variables when generating test cases.

One way in which function variables could be tested is if the user specified all non local variables within the function that determine the return value in the design document and value combinations are generated for these variables. Another approach would be to augment the system source code to replace the call to the function by another variable whose values can be set for testing purposes.

[3] "SD" and "SO" are the abbreviated names of "StockDB" and "StockOrders" respectively, and *random* indicates that the value is a generated randomly.

Interaction with external entities: Ideally the test cases should be run in a controlled environment such that any errors may be isolated to the unit under test. As a plan of an agent executes, it may interact with other entities, external to the agent containing the units being tested. We deal with two types of such external interactions: interactions with (i) external systems (e.g. external database server or another agent system) or with (ii) external agents that are part of the same system.

With respect to external systems, the interface to the agent under test may take various different forms, hence it is not straightforward to simulate this interaction in a controlled manner. Also, under the assumption that the external system is "fixed" (i.e. it is not under development, or able to be influenced), it is important that the unit under test respond appropriately to any interaction that happens with regard to such a system. Therefore the user is expected to ensure that such systems are accessible and functioning correctly prior to start of testing. User alerts/notifications to check this are generated based on design documentation, but can be turned off if desired.

If the interaction is with another agent of the same system, the form of interaction is known, making it possible to simulate and control this interaction. This is particularly important as some of these *interactee agents* (i.e. the external agents that the plan interacts with) may not have been fully implemented or tested. The technique employed is to replace the interactee agent with a test stub [8, p.148], [13, p.963]. We call such a test stub as a *Mock Agent*.

Functionality of a Mock Agent: A mock agent simulates the message send-reply logic of the interactee agent that it replaces. When a plan is executed during its testing process, any message from the plan to an interactee agent will be received by the replacement mock agent. When the mock agent receives a message from the plan-under-test, it will have one of two possible responses dependent on the design specification: (i) If the design does not specify a reply to the message received, the mock agent will just log the message received, and do nothing else; (ii) If the design specifies a reply message to the message received, the mock agent will log the message received, generate a reply message and send it to the plan-under-test. The data contained in the reply message in the current implementation is randomly assigned as the mock agent only simulates the interactee agent at the interaction level, and does not realise its internal logic[4].

Implementing a Mock Agent: The mock agents are automatically created when the testing framework builds the testing code for a plan. The process for generating the code of mock agents for a plan is as follows. First, all outgoing messaged are extracted from the plan under test. For each message, the interactee agent type that receives the message is identified. For each of the identified interactee agent type, the testing framework generates the code of a mock agent type that replaces this interactee agent type, following the rules below:

– The mock agent type shares the same type name as its replaced interactee agent type. Furthermore, if the interactee agent type has been implemented, the mock agent type also implements the same constructors as the constructors of it.

[4] We are in the process of allowing users to define specific responses associated with certain test cases, in a similar manner to other user defined test case information.

- For each message received by the interactee agent type, one plan is defined that handles this message in the mock agent. If this message has a reply specified in the design, this newly defined plan will create an object of the reply message and send it back to the agent-under-test. Else, the plan simply logs the message received.
- The code of this mock agent type is embedded into the test implementation to replace the code of its respective interactee agent type. Any message that is sent to the interactee agent will be received by this mock agent.

5 Implementation

The testing tool and approach described has been implemented within PDT, relying on the implemented agent system being in JACK. The implementation which does code augmentation for testing also relies on the code having a particular structure, and so requires that code skeletons are generated via PDT. The design descriptors used are those that are part of the normal design process. However, additional testing descriptors have been added, in order to map design variables to implementation variables, and to specify details of initialization that is necessary for execution.

An initial validation of the testing tool has been done using the case study of the *Electronic Bookstore* system as described in [14]. The *Stock Manager* agent was implemented according to the design specified, and then deliberately seeded with a range of faults, such as failing to post an expected message. For example, the *Stock Manager* agent is designed with the plan *Out of stock response* which posts the subtask *Decide supplier*, if the default supplier is out of stock and the number of books ordered is not greater than 100. The code was modified so that the condition check never returned true, resulting in the *Decide supplier* subtask never being posted. Examples of all faults which the testing system would be expected to recognize, were introduced into the *Stock Manager* agent. The testing framework generated 252 test cases using the "Normal" level of value generation and "Extended" combinations of values, as described in section 3. As could be expected, all seeded faults were found by the tool.

We are currently in the process of doing a more thorough evaluation, using a number of programs with design models, developed as part of a class on Agent Programming and Design. Once this is completed we will be able to comment on the kind of faults identified and the relationship of these to errors identified during marking the assignments. We have also evaluated the testing system on a demonstration program developed for teaching purposes. In this program 22 units were tested with 208 automatically generated test cases. This uncovered 2 errors where messages were not posted as specified in the design (apparently due to unfinished coding), and one error where invalid data caused an exception to be thrown. This program was developed as an exemplar and had been tested by its developer. Consequently the results of this testing process do indicate that the generation and execution of automated test cases, based on the design models, was effective in uncovering bugs in the system.

6 Discussion and Conclusion

In this paper we have presented an approach to testing of agent systems, which relies on unit testing with extensive coverage, and on design models as input to the analysis

of correct vs faulty behaviour. This work is based on the testing framework described in our previous work [3], where, we define the units to be tested, the scope of testing (the type of faults) and outline the testing process, providing some detailed mechanisms such as the algorithms for determining the order in which the units are to be tested.

In this work we provide the specific details of how variables are specified, extracted and assigned values to create test cases, and mechanisms for setting up the environment to execute the test cases automatically.

The approach presented is an unit testing approach to goal-plan based agent systems. The approach and mechanisms are implemented in PDT and JACK, and hence some of the examples that we have illustrated are PDT/JACK specific. However, we note that the approach, concepts, algorithms and processes can be adopted to any goal-plan based agent system. The design tool should allow for the information required to be specified and design time and automatically extracted at the testing phase. The code augmentation and monitoring would need to be implemented in the agent programming language as we have done in JACK following the algorithms and processes detailed in this paper and our previous work [3].

Different approaches to model based testing have focused on different kinds of models, which are then used to generate certain kinds of test cases. For example Apfelbaum and Doyle [15] describe model based testing focusing on use case scenarios defined by sequences of actions and paths through the code, which are then used to generate the test cases. This kind of testing is similar to integration testing or acceptance testing, and is something we would eventually hope to add to our system. Others (e.g. [16]) focus on models that specify correct input and output data, but these are not so appropriate for testing of complex behaviour models.

Binder [13, p.111] has summarized the elements of UML diagrams, exploring how these elements can be used for test design and how to develop UML models with sufficient information to produce test cases. He developed a range of testability extensions for each kind of UML diagram where such is needed for test generation. What we have done is somewhat similar to this for the basic elements of agent systems.

Besides Prometheus, various agent system development methodologies such as Tropos [17], MASE [18] and others, also have well developed structured models that could be used in a similar way to the PDT models used in this work. Several agent platforms do already use design models for some level of assistance in generation of test cases.

For example, the eCAT system associated with Tropos [19,20,21,22] is a testing tool that applies a goal oriented testing approach, which is based on the goal model of an agent based system. Such a goal model specifies the goal hierarchy of the system, describing how goals are decomposed to elementary goals. It is supplemented with ontology information for the purpose of automated test input generation [22], Positive test cases and negative test cases are then derived from the goal model to respectively verify the fulfillment (or not) of every elementary goal. However, the eCAT system does not explore internal components within an agent such as events and plans and does not verify if they work as expected.

Knublauch [23] introduces a set of APIs that are extended from the JUNIT testing framework [24], and explores an approach to test agent systems based on the design model of the Gaia methodology using these APIs. Human developers can use these

APIs to develop test cases that are automatically executed. However, test cases are still required to be manually generated. Also, this approach does not explore agents' internal units.

Other agent development systems also have their testing support subsystems. Coelho et.al [25] develop the JAT framework for testing the agents developed by the JADE [26] platform. They have defined a fault model that specifies a set of fault types relevant to general agent features. The testing framework provides skeleton code for human developer to manually develop test cases based on the fault model. A *tester agent* is used in the testing framework to automate the execution of test cases. Tiryaki develops the SUNIT [27] testing framework, which tests the agent systems developed by the Seagent platform [28]. SUNIT, which is also on top of JUNIT, specifies a set of APIs for human developers to manually generate test cases based on the SEAGENT model of an agent system. The SUNIT framework also explores internal features of an agent, such as the plan-level structure hierarchy of an agent and actions performed by plans.

To our knowledge, our testing tool is the only agent testing system which focuses on unit testing internal components of an agent, and also fully automates the generation of test cases as well as the running of them.

We believe that comprehensive automated unit testing is a critical first step in thorough testing of agent systems. However it is limited in what it tests for. Yet, the fact that the tool managed to find errors in a carefully developed and manually tested system, is an indication that the approach followed is already valuable.

In future work we expect to add testing related to the satisfactory following of scenarios as specified during requirements, as well as testing of agent interactions. We would expect to be able to adapt the debugging work of [6] using protocol specifications for the agent interaction testing and also refer to other research on interactions in agent systems (such as [29]).

References

1. Benfield, S.S., Hendrickson, J., Galanti, D.: Making a strong business case for multiagent technology. In: Proceedings of AAMAS 2006, pp. 10–15. ACM, New York (2006)
2. Padgham, L., Winikoff, M.: Developing Intelligent Agent Systems: A Practical Guide. John Wiley and Sons, Chichester (2004)
3. Zhang, Z., Thangarajah, J., Padgham, L.: Automated Unit Testing For Agent Systems. In: 2nd International Working Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2007), Spain, pp. 10–18 (July 2007)
4. Rao, A.S., Georgeff, M.P.: BDI Agents: From Theory to Practice. In: Lesser, V. (ed.) The First International Conference on Multi-Agent Systems, San Francisco, pp. 312–319 (1995)
5. Busetta, P., Rönnquist, R., Hodgson, A., Lucas, A.: JACK Intelligent Agents — Components for Intelligent Agents in Java. AgentLink News (2) (1999)
6. Poutakidis, D., Padgham, L., Winikoff, M.: An Exploration of Bugs and Debugging in Multi-agent Systems. In: Proceedings of AAMAS 2003, pp. 1100–1101. ACM, NY (2003)
7. Paton, N.W., Díaz, O.: Active Database Systems. ACM Comput. Surv. 31(1), 63–103 (1999)
8. Burnstein, I.: Practical Software Testing. Springer, New York (2002)
9. Jorgensen, P.C.: Software Testing: A Craftsman's Approach, 2nd edn. CRC Press, Boca Raton (2002)

10. Copeland, L.: A Practitioner's Guide to Software Test Design. Artech House, Inc., Norwood (2003)
11. Myers, G.J., Sandler, C., Badgett, T., Thomas, T.M.: The Art of Software Testing, 2nd edn. Wiley, Chichester (2004)
12. Cohen, D.M., Dalal, S.R., Fredman, M.L., Patton, G.C.: The AETG system: An Approach to Testing Based on Combinatiorial Design. Software Engineering 23(7), 437–444 (1997)
13. Binder, R.V.: Testing Object-Oriented Systems: Models, Patterns, and Tools. Addison-Wesley Longman Publishing Co., Inc., Boston (1999)
14. Padgham, L., Winikoff, M.: Developing Intelligent Agent Systems: A practical guide. John Wiley and Sons, Chichester (2004)
15. Apfelbaum, L., Doyle, J.: Model Based Testing. In: The 10th International Software Quality Week Conference, CA, USA (1997)
16. Dalal, S.R., Jain, A., Karunanithi, N., Leaton, J.M., Lott, C.M., Bellcore, B.: Model-Based Testing in Practice. In: International Conference on Software Engineering (1999)
17. Bresciani, P., Perini, A., Giorgini, P., Giunchiglia, F., Mylopoulos, J.: Tropos: An Agent-Oriented Software Development Methodology. Autonomous Agents and Multi-Agent Systems 8(3), 203–236 (2004)
18. DeLoach, S.A.: Analysis and Design using MaSE and agentTool. In: Proceedings of the 12th Midwest Artificial Intelligence and Cognitive Science Conference, Oxford, Ohio (2001)
19. Nguyen, C.D., Perini, A., Tonella, P.: A goal-oriented software testing methodology. Technical report, ITC-irst (2006), http://sra.itc.it/images/sepapers/gost-techreport.pdf
20. Nguyen, C.D., Perini, A., Tonella, P.: Automated continuous testing of multi-agent systems (December 2007)
21. Nguyen, C.D., Perini, A., Tonella, P.: Experimental evaluation of ontology-based test generation for multi-agent systems, pp. 187–198 (2009)
22. Nguyen, C.D., Perini, A., Tonella, P.: Ontology-based Test Generation for Multi-agent Systems. In: Proceedings of AAMAS 2008, pp. 1315–1320 (2008)
23. Knublauch, H.: Extreme programming of multi-agent systems (2002)
24. Husted, T., Massol, V.: JUnit in Action. Manning Publications Co. (2003)
25. Coelho, R., Kulesza, U., von Staa, A., Lucena, C.: Unit Testing in Multi-Agent Systems using Mock Agents and Aspects. In: Proceedings of the 2006 International Workshop on Software Engineering for Large-Scale Multi-Agent Systems, pp. 83–90 (2006)
26. Bellifemine, F., Poggi, A., Rimassa, G.: JADE: a FIPA2000 Compliant Agent Development Environment. In: Proceedings of Agents Fifth International Conference on Autonomous Agents, pp. 216–217 (2001)
27. Tiryaki, A.M., Öztuna, S., Dikenelli, O., Erdur, R.C.: SUNIT: A unit testing framework for test driven development of multi-agent systems. In: Padgham, L., Zambonelli, F. (eds.) AOSE VII / AOSE 2006. LNCS, vol. 4405, pp. 156–173. Springer, Heidelberg (2007)
28. Dikenelli, O.: SEAGENT MAS Platform Development Environment. In: Proceedings of AAMAS 2008, pp. 1671–1672 (2008)
29. EI Fallah-Seghrouchni, A., Haddad, S., Mazouzi, H.: A formal study of interactions in multi-agent systems. In: Garijo, F.J., Boman, M. (eds.) MAAMAW 1999. LNCS, vol. 1647. Springer, Heidelberg (1999)

Qualitative Modeling of MAS Dynamics

Using Systemic Modeling to Examine the Intended and Unintended Consequences of Agent Coaction

Jan Sudeikat* and Wolfgang Renz

Multimedia Systems Laboratory,
Hamburg University of Applied Sciences,
Berliner Tor 7, 20099 Hamburg, Germany
Tel.: +49-40-42875-8304
{jan.sudeikat,wolfgang.renz}@haw-hamburg.de

Abstract. The design of agent-based software applications is supported by modeling approaches that focus on the design of individual agents as well as their arrangement in organizational structures. However, it is a challenging task to deduce the collective system behavior from the sum of designs of the autonomous, pro-active actors and their collaboration can lead to surprising effects. Therefore, developers of agent collectives require tools to plan for the effects of agent coaction. Here, we propose a *systemic* modeling level that supplements established agent-oriented modeling approaches and utilizes *System Science* modeling concepts to support *qualitative* examinations of the system-wide dynamics that can result from agent coaction. We discuss the systematic derivation of systemic MAS abstractions from established design notations and exemplify systemic modeling of MAS in a case study where the interdependencies of agent activites and their consequences are identified at design-time and compared to run-time effects.

1 Introduction

The development of *Multi-Agent Systems* (MAS) is supported by sophisticated modeling approaches and development methodologies [1] that facilitate the design of software systems as sets of autonomous, pro-active agents. These tools typically focus on the conception of agent types and agent societies. The collaboration and interaction of agents establish the intended system functionality and the effective coordination of agents is a crucial development effort.

However, the consideration of collective effects in agent-based application designs is largely unexplored in methodical development procedures. In the natural sciences it has been noticed that *more is different* [2], i.e. that increasing

* Jan Sudeikat is doctoral candidate at the Distributed Systems and Information Systems (VSIS) group, Department of Informatics, Faculty of Mathematics, Informatics and Natural Sciences, University of Hamburg, Vogt–Kölln–Str. 30, 22527 Hamburg, Germany, jan.sudeika@informatik.uni-hamburg.de

M.-P. Gleizes and J.J. Gomez-Sanz (Eds.): AOSE 2009, LNCS 6038, pp. 80–93, 2011.

the quantity of equal system elements can lead to qualitatively different system behaviors. Examples are *self-organized* and *emergent* phenomena where global structures arise from the local interactions of individuals (e.g. particles, cells, etc.). Similar effects have been observed in distributed software systems [3] and the structural affinity between *complex systems* and MAS has been discussed [4] and observed [5]. Due to this affinity, it can make a qualitative difference whether agents (inter-)act in small or large populations and small redesigns of agent models can have unforeseen consequences.

Therefore, it is not sufficient for methodical development practices to check whether system elements behave as specified, but it is also necessary to check if the coaction of the conceived elements is able to meet application requirements. While the latter typically requires macroscopic system simulations (cf. section 5), here we discuss the examination of dynamic system properties at *design-time*. Particularly, we show how a *systemic* modeling level, that takes inspiration from *System Dynamics* [6], can be applied to anticipate the qualitative, macroscopic behavior of MAS. The systematic derivation of these models from MAS designs allows developers to inspect the causal structure of the application designs and facilitates examining the dynamic properties, e.g. the presence of oscillations and fixed points.

This paper is structured as follows. In the coming section, the systemic modeling of MAS is discussed. In section 3, a procedure is presented that guides the analysis of the dynamic properties of MAS designs. The systematic derivation of systemic MAS models as well as their examination are discussed and exemplified in section 4. Finally, we conclude and give prospects for future work.

2 Systemic Modeling of Collective Agent Behavior

System Dynamics provides an interdisciplinary approach to model dynamical systems [6] and anticipate their timely behavior. At any given time-point the system state is given by a set of accumulative values of *system variables*. These variables influence each other mutually, i.e. *causal relations* denote the additive (positive) or subtractive (negative) rates of changes of these properties. Circular structures form feedback loops that are either *balancing* (-) and damp variable fluctuations or *reinforcing* (+) and amplify perturbations of system variables. An established formalism is the *Causal Loop Diagram* (CLD), a graph-based notation which denotes system properties as linked nodes [6].

System dynamics and agent-based modeling provide contradicting approaches to understand and simulate systems (e.g. cf. [7]). However, it has been found that the explicit modeling of (causal) feedback structures within MAS is applicable to describe the requirements on adaptive MAS [8], organizational dynamics [9] and decentral coordination schemes [10]. This modeling level is applicable when the causal interdependencies among agent types are known at design time. It can be used to examine collective effects that originate from non-linear agent interactons as well as MAS where multiple agent types can exhibits the same behaviors/roles.

Here, we outline a MAS-specific refinement of CLD variable and link types that has been given in [11] to facilitate the unambiguous modeling of MAS (see [12] for a discussion of complications). Methodology and agent-architecture independent descriptions of macroscopic MAS states are utilizing the generic *role* and *group* concepts. *Roles* characterize agent activities, i.e. commitments to normative agent behaviors, and *groups* provide the context of roles by partitioning MAS organizations into sets of individuals that share characteristics [13]. An *Agent Causal Behavior Graph* (ACBG) is denoted by: $ACBG :=< V_{acbg}, E_{acbg} >$, where V_{acbg} is a set of nodes and E_{acbg} is a set of directed edges between nodes ($E_{acbg} \subseteq V_{acbg} \times V_{acbg}$). Nodes indicate system variables that denote the number of agents that exhibit observable behaviors (e.g. cf. [14]). Different types of nodes and the corresponding variables denote the number of current *role activations* ($r_{(x)}$), the number of *active groups* ($g_{(x)}$), the group *size* ($gs_{(x)}$) and the cumulative values of *environment properties* ($e_{(x)}$). An additional node type can be used to join links to a combined interaction *rate* ($r_{(x)}$) that expresses contributive influences which jointly cause behavior adjustments of agents. Links denote *positive* or *negative* causal influences that are either *direct* ($e_{(d+/-)}$), e.g. due to inter-agent communication, or *mediated* ($e_{(m+/-)}$) by coordination media, e.g. shared environments [15]. Links between nodes of MAS design elements ($r_{(x)}$,$g_{(x)}$,$gs_{(x)}$,$e_{(x)}$) describe *causal* relations. Therefore, the increases of a node value enforces that the values of positively connected node values increase in subsequent time steps. Negative relations enforce changes in opposite directions [6]. Rate-nodes have only additive / subtractive influences on connected nodes. Details on the graphical (cf. figure 7) and textual modeling of ACBGs can be found in [9,10].

3 Systematic Examination of Qualitative MAS Dynamics

Here, we outline a procedure to examine the dynamic properties of agent-based application designs. Developers abstract design models to describe the conceived application as a dynamical system and apply simulation tools to examine the space of possible system behaviors. These tasks supplement established development practices (e.g. reviewed in [1]) by an optional development activity, i.e. a method fragment [16]. This extension enables developers to anticipate the collective effects of agent models at analysis and design phases in MAS development.

Figure 1 outlines the *MAS Coordination Analysis* Activity[1]. This Activity comprises three phases, namely the (pre-)processing of design models (A), the construction of an ACBG (cf. section 2; B) and its analysis (C).

3.1 Design Preprocessing

Initially, MAS design models, which are specific to certain methodologies and/or agent architectures, are abstracted to extract the information that are required

[1] Following OMGs *Software Process Engineering Meta-Model* (SPEM) notation: `http://www.omg.org/technology/documents/formal/spem.htm`

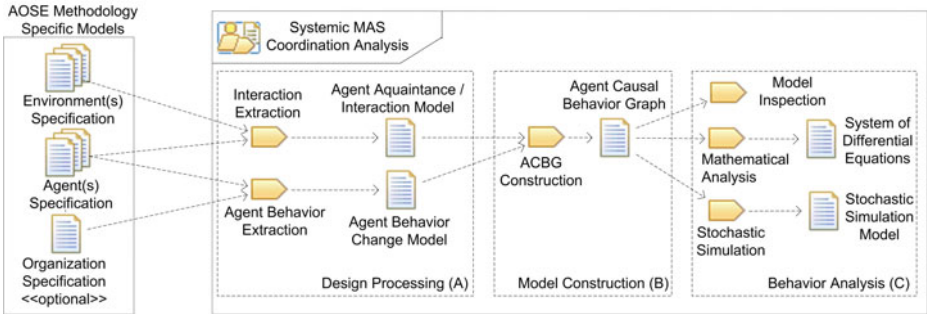


Fig. 1. The qualitative examination of MAS Dynamics

to derive ACBGs (cf. figure 1; A). Three types of design models are considered, namely specifications of the agent(s), of their environment(s) and optional models of organizational structures. The *Agent Behavior Extraction* addresses the derivation of observable agent behaviors, i.e. ACBG nodes, and the *Interaction Extraction* addresses the extraction of the interdependencies between the agent(s) as well as environment properties. The proposed Activity can be adjusted to different modeling formalisms by providing extraction guidelines.

Figure 2 denotes the conceptual models of the extracted data structures. An *Agent Interaction* (AI) model (cf. figure 2; left) describes the interdependencies (*Inter-Agent Interaction*) of *Agent* types and *Environment* elements. Some methodologies include comparable models, e.g. the GAIA *acquaintance model* [17]. Direct, message-based interactions are commonly denoted by modeling tools, e.g. as *Protocols* in *Prometheus* (*System Overview Diagrams*) [18]. In addition, developers have to search designs for mutual interactions with environment elements, as these may indicate mediated agent interdependencies [19].

The construction of an AI model, i.e. the Task *Interaction Extraction*, is denoted in Figure 3 as a stereotyped UML 2 Activity Diagram. In agreement with [20], Tasks are described as a control flow that governs the execution of *Steps*. The

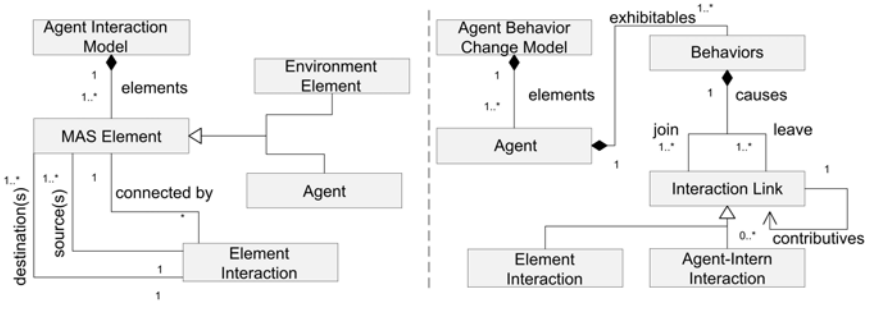


Fig. 2. Conceptual models of intermediate models that prepare ACBG construction

illustrated processing comprises two principal phases. First, MAS design models are searched for interacting types of system elements. In two distinct steps the *active* (*Identify Actors / Agents*), i.e. agents and system actors, as well as *passive* (*Identify Environment Elements*) types of system element are identified.

In the second phase, the interactions of the identified elements are extracted. In parallel, the MAS design is searched for three types of *direct* interactions (*Identify Direct Interactions:*). First, interactions among sets of active system elements are considered (*Agent/Agent*). These are signified by message exchanges that are possibly structured by interaction protocols. Secondly, the interactions are examined that involve passive system elements (*Agent / Environment*). These describe the modification of environment elements as well as the sensing of these modifications. Finally, the interactions and infleunces between passive environment elements are considered (*Environment/Environment*). The interactions that are identified in the latter two steps serve as input to the identification of *mediated* interactions (*Infer Mediated Interactions: Agent/Agent*). In this step, the mutual acting/sensing activities of agents are related to each other by infering infomration flws that are manifested by transient modifications of environment elements. The popagation of modifications is traced to identify which agents can be affected by the modification of environment elements.

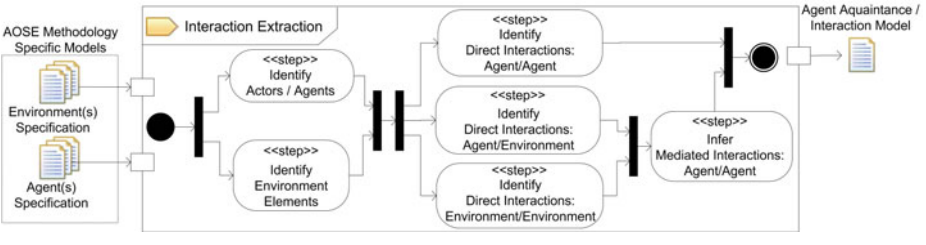


Fig. 3. Procedure to extract the Agent Interaction model

An *Agent Behavior Change* (ABC) model (cf. figure 2; right) describes the behaviors that individual agent types can exhibit and denotes the influences that cause agents to adjust their local behaviors [21]. The model consists of a set of *Agent* representations that comprise a list of observable *Behaviors*. Behavior representations denote the causes, i.e. the *Interaction Links*, that make individual agents *join* and *leave* behaviors. These causes of behavior changes relate to interactions between agents / environment elements (*inter-agent*) or internal events, e.g. goal adoptions or periodic activations (*agent-intern*). Inter-agent interactions are identified in the AI models.

Figure 4 summarizes the Task *Agent Behavior Extraction* that adresses the construction of an ABC model. The Step *Identify Behaviors* unambiguously identifies agent behaviors in agent specifications. Particularly duplicates are resolved, i.e. behaviors that are denoted in both design models. In [22], this

identification is exemplified for deliberative, goal-directed agents where agent behaviors can be distinguished by iterating the static tree structure of goals, (sub-)goals and plan implementations. The level of detail of the behavior identification controls the granularity of the analysis [22]. The observable behaviors are identified for each agent and for each behavior the causes to adopt and drop it are identified (*Search Causes*). In this respect denote causes the interactions that make agents adjust their behavior. These describe not necessarily imperatively, reactive responses but also denote the events that make the agent reasoning to consider adoption/leave of a behavior. These causes are added to the ABC model (*Add Interaction*), if not already present.

For the causing interdependencies it is checked whether they solely cause behavior adjustments or they *contribute* (*is not contributive*), i.e. that interactions conjointly enforce adjustments (cf. section 2). An example is given in section 4.2, where the activation of agents requires the availability of inactive agents as well as environment elements. The conjoint interactions are added for later processing (*Add Contributive Interactions*). When all causes are processed, the behavior is added to the ABC model (*Add Behavior*). The Task is finished when the set of behaviors, exhibited by all active system elements, has been processed. Organizational modeling formalisms describe when agents join/leave groups and roles, e.g. in the Agent-Group-Role (AGR) model AUML sequence diagrams are utilized for this purpose [23]. These descriptions denote the interactions / perceptions that cause behavior changes. If an organizational model of the MAS is available [13], this is also searched as well for agent behaviors, i.e. role and group concepts (cf. section 2), and the identified influences are processed as described above.

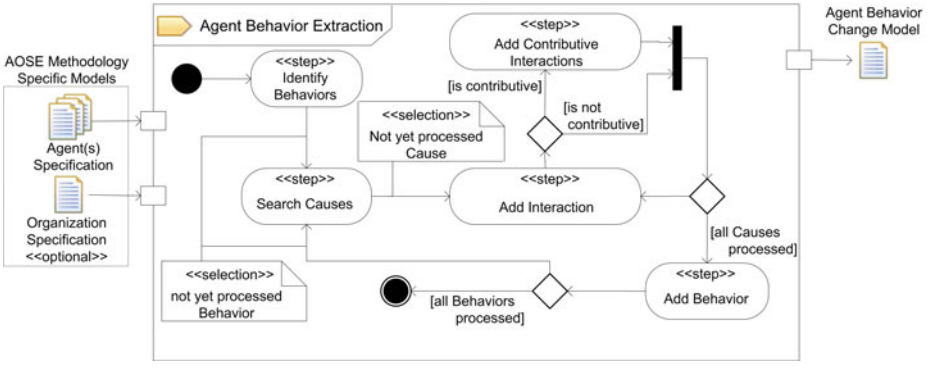


Fig. 4. Procedures to the processing of MAS designs

3.2 Systemic Model Construction

The (pre-)processing of MAS designs leads to intermediate data structures, i.e. AI and ABC models. These are processed in the Task *ACBG Construction* (cf. Figure 5) to derive a the systemic system representation. First, models are processed to identify the ACBG nodes and add these to the ACBG model (*Add*

ACBG Nodes (ABCM / AIM)). In this step, agent behaviors, denoted in the ABC Model are mapped to *role activation* ($r_{(x)}$), *active groups* ($g_{(x)}$) or *group size* ($gs_{(x)}$) node types (cf. section 2). In addition, environment elements (from the the AI model) are mapped to ACBG *environment property* nodes ($e_{(x)}$).

Then the links are introduced. For each node the corresponding causes (ABC model) are iterated. If the interaction is not present in the graph, it is introduced (*Add ACBG Edge*) and connected (*Connect Nodes*). These links describe positive (negative) relations when they cause equal (opposite) directed variable changes, e.g. an increase (decrease) causes agents to adopt connected roles. Links may also be contributive (*isContributive*), i.e. combinations of influences cause changes of behaviors and/or environment elements. Contributive links are joint by connecting these to a common *rate* node ($r_{(x)}$) and connecting the rate to the target nodes of the interactions (*Add Contributives + Join Edges*). The contributions of joint interactions is expressed by their individual infleunces on a global adjustment rate. This rate characterizes how agent interactions infleunce behavior adjustments. The Task is finished when the set of nodes is processed and for each node the identified causes have been addedd to the ACBG model. This procedure does not check whether the introduced graph nodes are linked, since orphaned nodes / agent types may indicate negligences in the application design.

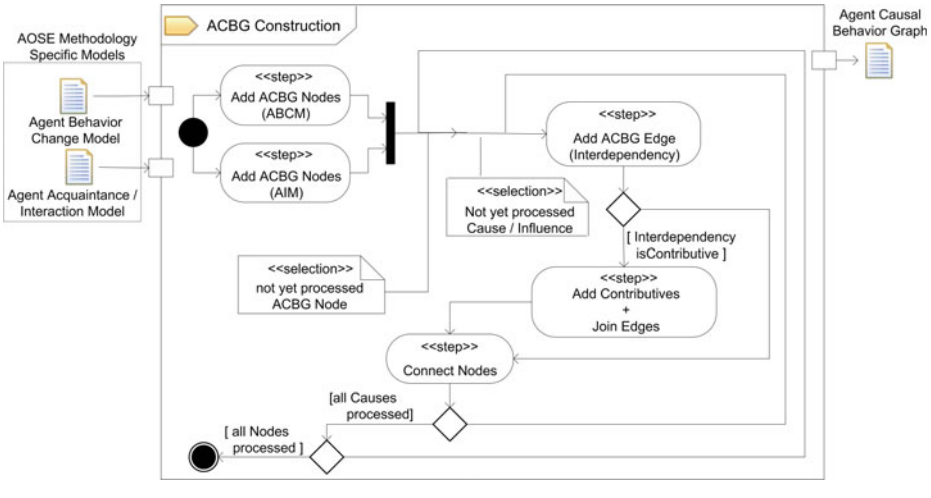


Fig. 5. ACBG construction procedure

3.3 Behavior Analysis

The structure of the derived ACBGs highlights the causalities between agent types and their behaviors. This viewpoint particularly eases the identification of cyclic causalities which manifest feedback loops. Developers will not only inspect

the model manually (cf. figure 1; *Model Inspection*), but also animate the system behavior to visualize the time dependent changes of system variables, e.g. to answer *what-if* questions on different parameterizations and initial conditions.

The animation requires that the CLDs (ACBGs, respectively) are manually transferred to more precise mathematical models and are calibrated to realistic configurations. Ordinary Differential Equations (ODE) are commonly used to denote the rates of change of system variables and can be iterated by numerical computing environments. The formulation of the system as a set of ODE enables a formal treatment (*Mathematical Analysis*, cf. figure 1), e.g. the stability analysis of fixed points [24]. An associated modeling formalism is the *Stock-and-Flow* Diagram [6]. This modeling approach expresses system variables as stocks and denote the quantitative in- and out-*flow* in / from these stocks with mathematical formulas. Simulation tools to iterate these models are freely available[2]. In addition, we found that stochastic simulation tools, e.g. stochastic process algebra [25] as utilized in [26], are appropriate to simulate ACBG models (cf. figure 1: *Stochastic Simulation*). System variables are represented by numbers of active processes and their activation / deactivations is controlled by stochastic interactions via interaction channels. The utilization of a Stock-and-Flow Diagrams is exemplified in section 4 and details on the stochastic simulation of ACBGs can be found in [27,9].

4 Case Study: Examining Marsworld Dynamics

In the following, the proposed analysis procedure is exemplified by examining the collective behavior of a comparatively simple MAS. The derivation and analysis of the qualitative dynamics of these models identifies causal interdependencies between agent activities and we show the agreement of the anticipated causalities with MAS simulation results.

4.1 The Marsworld Design

The so-called *Marsworld* follows a case study given in [28]. The following description is based on an interpretation of this scenario that is freely available as an example application of the *Jadex* agent system.[3] The hypothetical *Marsworld* setting addresses mining activities on a far distant planet. Teams of autonomous robots are responsible to *explore*, *mine* and *transport* ore. *Sentry* agents are equipped with specific sensors that can verify ore locations. *Producer* agents are equipped with mining devices and *Transporter* agents can carry ore to the home base. Producers and Transporters are also equipped with inferior sensors that report locations that may accommodate ore. These locations are communicated to sentry agents. Sentry agents move to suspicious sites and verify the presence of ore. When ore has been located a randomly selected Producer agent is called

[2] E.g. the System Dynamics module of the Netlogo environment: `http://ccl.northwestern.edu/netlogo/`

[3] `http://vsis-www.informatik.uni-hamburg.de/projects/jadex/`

to mine the ore. When the location has been exhausted a randomly selected transporter agent is called to move the resource to the home base.

The agent models comprises two behaviors. Agents either search or execute their designated activity, i.e. sense, produce or transport. Figure 6 (left) denotes the logical structure of the MAS design and details the Sentry agent type in the Tropos [29] notation. All agents are equipped with a default behavior to search for ore. When ore locations are spotted, these are communicated to Sentry agents that *verify ore* locations. This activity comprises to move to the suspicious location, to use the sensors and to request the production of the ore. In a similar way, Producer and Transporter agents provide their designated activities and Transporters depend on Producers to communicate requests to transport ore.

Figure 6 (right) describes the activation of Sentry agents (B.1) and Producer agents (B.2) by AUML sequence diagrams [30]. Any agent type is subject to the perception of ore locations. This causes the communication of the location (*message: inform location*) to sentry agents and causes them to adjust their behavior, i.e. start the sensing activity. When the sensing activity is finished, Sentry agents send requests to produce the identified ore to producer agents (message: request production). This causes Sentry agents to enter the searching behavior and makes the receiving Producer agent to start production.

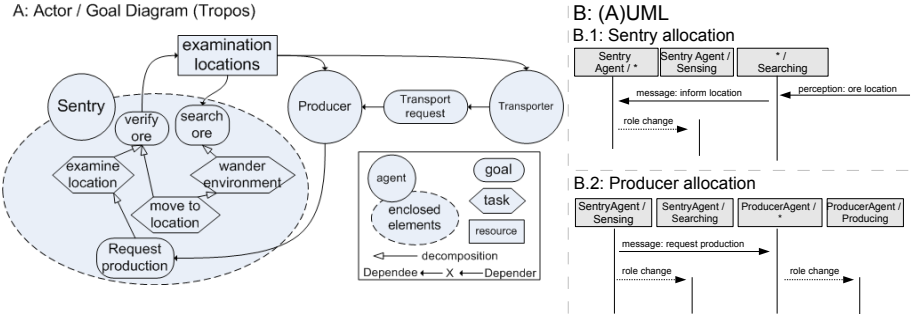


Fig. 6. A Tropos Actor (Goal Diagram) that denotes the internal structure of the Sentry agent type (left) and (A)UML sequence diagrams (right) of the agent interactions

4.2 Examining the Marsworld Dynamics

The derivation of an ACBG from the outlined MAS design highlights its causal structure and reveals feedback loops (cf. figure 7). The agent types are each described by two role nodes that denote the numbers of activated and searching agents. The sum of these nodes describes the MAS state. The state of the environment is given by the numbers of locations that are subject to the activity of Sentry agents, Producer agents and Transporter agents. When the searching Producer and Transporter agents encounter unexplored *Resources*, the number of Sentry activations increases. Therefore, the search and random encounter of Resources are contributive interactions (cf. section 2) that influence the rates of

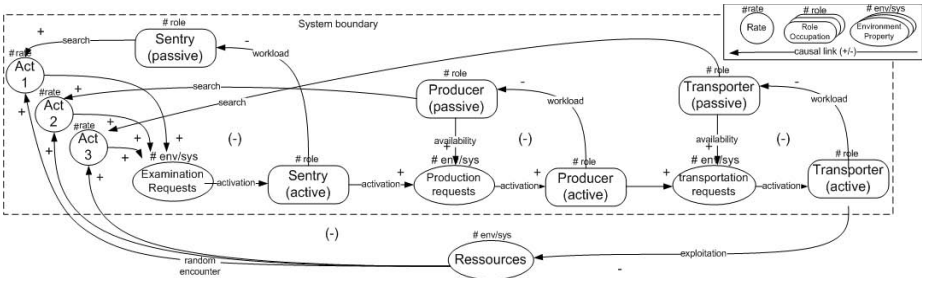


Fig. 7. Systemic Model (ACBG, cf. section 2) of the Marsworld dynamics

sentry activations. Finally, the activity of Transporters is reducing the number of available Resources (negative link). Therefore, the MAS establishes a balancing feedback loop that continuously removes Resources from the environment. The activation of Sentries is influenced by three balancing feedbacks. As the activations of sentries increases, the activations of the other agent types increases as well (positive links). However, these activations of agents limits the number of searching agents in the system. Thus the activation rates of the Sentries are decreased as well. The sentry activations increase when less agents are active, i.e. more agents are occupied with searching the environment.

4.3 Discussion

We compare the derived ACBG (figure 7) with an agent-based Marsworld simulation[4]. By enforcing a stationary working regime, e.g. by a fixed input rate of resources, the interdependencies of agent activities can be observed as correlations of agent activities (cf. figure 8; C, D). The systemic models assume that agent cardinatlities are above one and ignores possible blocking by serialization constraints. The correlation of the active sentries with the active producers (cf. figure 8; C) shows a maximum at a delay time of about 60 time units. Since the curve is asymmetric, the mean delay time is somewhat longer (about 70). The same delay time is shown by the correlation of active producers with the active transporters. This correlation function has a much longer tail, since the transporters are activated for a longer time and move between ore-source and the home-base. Consequently, the correlation between active sentries and active transporters also has a long tail at a delay time being equal to the sum of the two above mean delay times. (In the figure, the opposite correlation with a negative time difference is plotted.) The auto-correlation functions (see figure 8; D) exhibit a symmetric decay with a short auto-correlation time of the active sentry and producer agents, and a much longer auto-correlation time of the transporters for the above explained reason. Interestingly enough, sentries have time slots of negative auto-correlation which show the interaction with the other non-active

[4] Using Netlogo: 100x Sentries, Producer, Transporter in a grid of 1225 patches.

agent roles. Thus, we have shown that even in a highly fluctuating stationary working regime, the system dynamics characteristics imposed by ACBG can be validated using correlation function techniques.

Figure 8 (B) animation of the ACBG that denotes the agent activations as relative scales. When teams encounter resources, the Sentries are activated (1), followed by the Producers and Transporters. Inactive agents search the environment and therefore contribute to the activation of Sentry agents (2). Figure 8 (A) shows this behavior in an agent-based simulation of a bounded environment that is initialized with a load of ore-sources that are reinforced as a low rate. The MAS exhibits a comparable system behavior, i.e. resources are foraged and afterwards agent activations approach a low steady state that is heavily perturbed by the spatial distribution of agents and resources. The systemic models particularly allow to examine specific system configurations (*what-if* games). Figure 8 (E, F) denote simulations of a MAS configuration, where the number of available Producers limits the system operation due to the possible activations of Transporters are limited as well. Therefore, agent activations approach steady states. This indicates the dependence of the effectivity of the MAS on the quantities of agents and the need for their adaptive allocation.

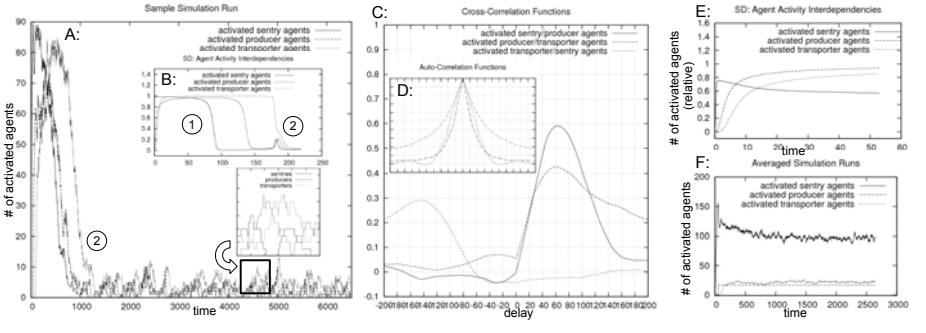


Fig. 8. Comparing System Dynamic and agent-based simulation models

5 Related Work

Refinements of causality graphs have been proposed to understand the intra-agent and inter-agent activities of MAS implementations. These approaches monitor agent execution and causally relate the observed events, e.g. in [31] to check agent interactions. In [32], graph structures denote the causal relations among agent concepts, e.g. that the reception of a message modifies a belief value. The generation of these graphs structures, by the monitoring of agents, faciltiates the comparison of the expectations on the agent execution with the actual behavior of system elements. These approaches focus on the microscopic behavior of individual agents while we are here addressing the macroscopic behavior that rises from collectives.

Mathematical modeling (e.g. [14]) and macroscopic system simulation (e.g. [26]) have been proposed to examine the dynamic properties, e.g. oscillations and fixed points, in MAS. In [14], reactive agents as well as environments are understood as stochastic processes that can be described by differential equations. This approach particularly addresses homogeneous MAS and the derived models are equivalent to the mathematical concretions that animate ACBG dynamics (cf. section 3.3). In [26], the collective behavior of agents is examined by translating them to stochastic simulation models, i.e. stochastic process algebra.

Here, we advocate the extraction and examination of the *systemic* structure of MAS prior to system simulations or mathematical iterations. The proposed modeling level utilizes *System Dynamics* concepts and the resulting models do not only expose collaborative effects but also facilitate their explanation. The observation of collaborative effects by model simulations/iterations requires parameter sweeps and the manual search for appropriate model calibrations is facilitated by insights in the causal structure that enable debelopers to infer the observable effects that can be expected.

6 Conclusions

In this paper, we argued that the systematic development of agent-based software systems has to plan for the collective effects that can rise from agent coaction. A *systemic* modeling level has been outlined that supplements current development practices with the ability to anticipate qualitative dynamic properties, i.e. oscillations and fixed points, which MAS architectures may exhibit. The systematic derivation and analysis of these models has been discussed and exemplified. Future work concerns the (semi-)automation of the presented procedure to analyse MAS designs. Systemic modeling was initially proposed to assist the design [8] and construction of self-organizing phenomena [11]. Therefore, it will be examined as well whether other approaches to design complex MAS behaviors, e.g. based on the AMAS theory [33], may benefit from this technique. In addition, the integration in established development processes will be examined in more detail. Besides the here presented *analytic* usage of systemic models, their *constructive* usage, e.g. to configure and enact externalized models of decentral coordination strategies, has been considered [10] and will be further developed.

Acknowledgment

One of us (J.S.) would like to thank the *Distributed Systems and Information Systems* (VSIS) group at Hamburg University, particularly Winfried Lamersdorf, Lars Braubach and Alexander Pokahr for discussion and encouragement.

References

1. Henderson-Sellers, B., Giorgini, P. (eds.): Agent-oriented Methodologies. Idea Group Publishing, USA (2005) ISBN: 1591405815
2. Anderson, P.: More is different. Science 177, 393–396 (1972)

3. Mogul, J.C.: Emergent (mis)behavior vs. complex software systems. Technical Report HPL-2006-2, HP Laboratories Palo Alto (2005)
4. Odell, J.: Agents and complex systems. Journal of Object Tech. 1, 35–45 (2002)
5. Parunak, H.V.D., Brueckner, S., Savit, R.: Universality in multi-agent systems. In: AAMAS 2004: Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems, pp. 930–937. IEEE Computer Society, Los Alamitos (2004)
6. Sterman, J.D.: Business Dynamics - Systems Thinking and Modeling for a Complex World. McGraw-Hill, New York (2000)
7. Van Dyke Parunak, H., Savit, R., Riolo, R.L.: Agent-based modeling vs. Equation-based modeling: A case study and users' guide. In: Sichman, J.S., Conte, R., Gilbert, N. (eds.) MABS 1998. LNCS (LNAI), vol. 1534, pp. 10–25. Springer, Heidelberg (1998)
8. Sudeikat, J., Renz, W.: On expressing and validating requirements for the adaptivity of self-organizing multi-agent systems. Sys. and Inf. Sci. N. 2, 14–19 (2007)
9. Renz, W., Sudeikat, J.: Modeling feedback within mas: A systemic approach to organizational dynamics. In: Proceedings of the International Workshop on Organised Adaptation in Multi–Agent Systems (2008)
10. Sudeikat, J., Renz, W.: MASDynamics: Toward systemic modeling of decentralized agent coordination. In: Proceedings of KIVS 2009 (2009)
11. Sudeikat, J., Renz, W.: Programming adaptivity by complementing agent function with agent coordination: A systemic programming model and development methodology integration. In: Proc. of the 5th Int. Conf. on Self-organization and Adaptation of Computing and Communications, SACC 2009 (2009)
12. Richardson, G.P.: Problems with causal-loop diagrams. Sys. Dyn. Rev. 2, 158–170 (1986)
13. Mao, X., Yu, E.: Organizational and social concepts in agent oriented software engineering. In: Odell, J.J., Giorgini, P., Müller, J.P. (eds.) AOSE 2004. LNCS, vol. 3382, pp. 1–15. Springer, Heidelberg (2005)
14. Lerman, K., Galstyan, A.: Automatically modeling group behavior of simple agents. In: Agent Modeling Workshop, AAMAS 2004, New York, NY (2004)
15. Gouaich, A., Michel, F.: Towards a unified view of the environment(s) within multi-agent systems. Informatica (Slovenia) 29, 423–432 (2005)
16. Cossentino, M., Gaglio, S., Garro, A., Seidita, V.: Method fragments for agent design methodologies: from standardisation to research. Int. J. Agent-Oriented Software Engineering 1, 91–121 (2007)
17. Zambonelli, F., Jennings, N., Wooldridge, M.: Developing multiagent systems: the gaia methodology. ACM Trans. on Software Eng. and Meth. 12, 317–370 (2003)
18. Padgham, L., Winikoff, M.: Developing Intelligent Agent Systems: A Practical Guide. John Wiley and Sons, Chichester (2004) ISBN 0-470-86120-7
19. Weyns, D., Brueckner, S.A., Demazeau, Y. (eds.): EEMMAS 2007. LNCS (LNAI), vol. 5049. Springer, Heidelberg (2008)
20. Object Management Group: Software & systems process engineering meta-model specification version 2.0, formal/2008-04-01 (2008), http://www.omg.org/spec/SPEM/2.0/PDF
21. Odell, J., van Dyke Parunak, H., Brueckner, S., Sauter, J.: Changing roles: Dynamic role assignment. Jounal of Object Technology 2, 77–86 (2003)
22. Sudeikat, J., Renz, W.: Monitoring group behavior in goal–directed agents using co–efficient plan observation. In: Padgham, L., Zambonelli, F. (eds.) AOSE VII / AOSE 2006. LNCS, vol. 4405, pp. 174–189. Springer, Heidelberg (2007)

23. Ferber, J., Gutknecht, O., Michel, F.: From agents to organizations: An organizational view of multi-agent systems. In: Giorgini, P., Müller, J.P., Odell, J.J. (eds.) AOSE 2003. LNCS, vol. 2935, pp. 214–230. Springer, Heidelberg (2004)
24. Kaplan, D., Glass, L.: Understanding Nonlinear Dynamics. Springer, Heidelberg (1995)
25. Priami, C.: Stochastic π–calculus. Computer Journal 6, 578–589 (1995)
26. Gardelli, L., Viroli, M., Omicini, A.: On the role of simulations in engineering self-organising MAS: The case of an intrusion detection system in tuCSoN. In: Brueckner, S.A., Di Marzo Serugendo, G., Hales, D., Zambonelli, F. (eds.) ESOA 2005. LNCS (LNAI), vol. 3910, pp. 153–166. Springer, Heidelberg (2006)
27. Sudeikat, J., Renz, W.: On simulations in mas development. In: Braun, T., Carle, G., Stiller, B. (eds.) KIVS 2007 Kommunikation in Verteilten Systemen – Industriebeitrge, Kurzbeitrge und Workshops. VDE-Verlag (2007)
28. Ferber, J.: Multi-Agent Systems. Addison-Wesley, Reading (1999)
29. Giorgini, P., Kolp, M., Castro, J.M.J.: Tropos: A Requirements-Driven Methodology for Agent-Oriented Software. In: Agent-Oriented Methodologies, pp. 20–45. IDEA Group Publishing, USA (2005)
30. Odell, J., Parunak, H.V.D., Bauer, B.: Extending uml for agents. In: Proc. of the Agent-Oriented Information Systems Workshop at the 17th National Conference on Artificial Intelligence, pp. 3–17 (2000)
31. Vigueras, G., Botia, J.A.: Tracking causality by visualization of multi-agent interactions using causality graphs. In: Dastani, M.M., El Fallah Seghrouchni, A., Ricci, A., Winikoff, M. (eds.) ProMAS 2007. LNCS (LNAI), vol. 4908, pp. 190–204. Springer, Heidelberg (2008)
32. Lam, D.N., Barber, K.S.: Comprehending agent software. In: Proceedings of the Fourth International Joint Conference on Autonomous Agents and Multiagent Systems, pp. 586–593. ACM Press, New York (2005)
33. Capera, D., George, J.P., Gleizes, M.P., Glize, P.: The amas theory for complex problem solving based on self-organizing cooperative agents. In: Enabling Technologies: Infrastructure for Collaborative Enterprises, WET ICE 2003, pp. 383–388 (2003)

Part III

Development Method Proposals

Methodology for Engineering Affective Social Applications

Derek J. Sollenberger and Munindar P. Singh

Department of Computer Science
North Carolina State University
Raleigh, NC 27695-8206, USA
{djsollen,singh}@ncsu.edu

Abstract. Affective applications are becoming increasingly mainstream in entertainment and education. Yet, current techniques for building such applications are limited, and the maintenance and use of affect is in essence handcrafted in each application. The Koko architecture describes middleware that reduces the burden of incorporating affect into applications, thereby enabling developers to concentrate on the functional and creative aspects of their applications. Further, Koko includes a methodology for creating affective social applications, called Koko-ASM. Specifically, it incorporates expressive communicative acts, and uses them to guide the design of an affective social application. With respect to agent-oriented software engineering, Koko contributes a methodology that incorporates expressives. The inclusion of expressives, which are largely ignored in conventional approaches, expands the scope of AOSE to affective applications.

1 Introduction

Representing and reasoning about affect is essential for producing believable characters and empathic interactions with users, both of which are necessary for effective agent based entertainment and education applications. Leading applications of interest include pedagogical tools [4,9], military training simulations [6], and educational games [10].

As evidenced by the above applications, incorporating affect into applications is an active area of research. The primary focus of the existing research has been modeling the affective state of a single agent. Our work builds on that foundation but goes further by incorporating affective computing with multiagent systems (MAS). By focusing our attention on the communication of affect between agents, we can develop a new class of applications that are both social and affective. To achieve the fusion of affect with MAS, two challenges must be overcome. First, we need a medium through which agents can exchange affective data. Second, we must define a methodology for creating affective social applications via that medium.

The first challenge is addressed by using Koko [14], a middleware that facilitates the sharing of affective data. Koko is a multiagent middleware whose agents manage the affective state of a user. Further, Koko is intended to be used by applications that seek to recognize emotion in human users. Although it is possible to use Koko in systems

M.-P. Gleizes and J.J. Gomez-Sanz (Eds.): AOSE 2009, LNCS 6038, pp. 97–109, 2011.

that model emotion in virtual characters, many of its benefits most naturally apply when human users are involved.

Importantly, Koko enables the development of affective social applications by introducing the notion of expressive communicative acts into agent-oriented software engineering (AOSE). Using the multiagent environment provided by Koko, agents are able to communicate affective information through the exchange of expressive messages. The communication of affective information is naturally represented as an *expressive* communicative act, as defined by Searle [12]. However, expressive acts are a novelty in both AOSE and virtual agent systems, which have traditionally focused on the assertive, directive, and commissive communicative acts (e.g., the FIPA inform command). Further, Koko enables us to create a methodology for engineering affective, social applications.

Contributions. This paper describes a methodology, Koko-ASM, centered on expressive communicative acts, the first such methodology to our knowledge. Using this methodology application developers can construct applications that are both social and affective. As such, the combination of the Koko middleware and this methodology enable AOSE to expand into the design and creation of affective social applications.

Paper Organization. The remainder of this paper is arranged as follows. Section 2 reviews appraisal theory affect models. Section 3 provides an overview of the Koko middleware. Section 4 describes the Koko-ASM methodology. Section 5 demonstrates its merits via a case study.

2 Background

This section provides a synopsis of two areas that are fundamental to Koko-ASM: (1) appraisal theory as a foundation for modeling affect and (2) communicative acts and their relevance to AOSE.

2.1 Appraisal Theory

Smith and Lazarus' [13] cognitive-motivational-emotive model, the baseline for current appraisal models (see Fig. 1), conceptualizes emotion in two stages: appraisal and coping. *Appraisal* refers to how an individual interprets or relates to the surrounding physical and social environment. An appraisal occurs whenever an event changes the environment as interpreted by the individual. The appraisal evaluates the change with respect to the individual's goals, resulting in changes to the individual's emotional state as well as physiological responses to the event. *Coping* is the consequent action of the individual to reconcile and maintain the environment based on past tendencies, current emotions, desired emotions, and physiological responses [8].

A situational construal combines the environment (facts about the world) and the internal state of the user (goals and beliefs) and produces the user's perception of the world, which then drives the appraisal and provides an appraisal outcome. This appraisal outcome is made up of multiple facets, but the central facet is *Affect* or current emotions. For practical purposes, *Affect* can be interpreted as a set of discrete states with an associated intensity. For instance, the result of an appraisal could be that you are simultaneously happy (at an intensity of α) as well as proud (at an intensity of β).

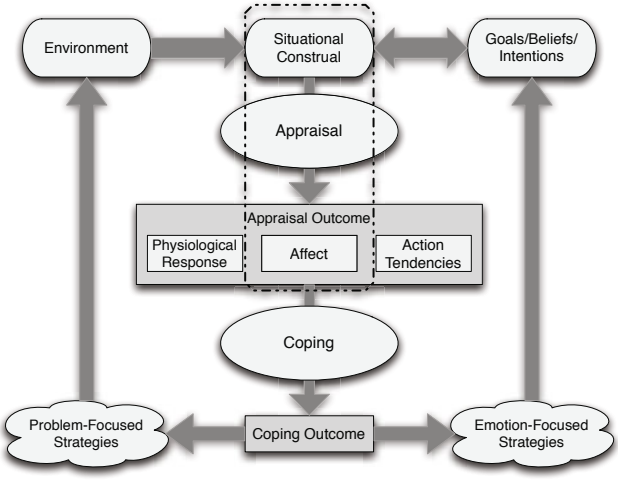


Fig. 1. Appraisal theory diagram [13]

2.2 Communicative Acts

The philosophers Austin [1] and Searle [12] developed speech act theory founded on the principle that communication is a form of action. In other words, when an agent communicates, it alters the state of the world. Communicative acts are grouped based on their effects on the agent's internal state or social relationships. Specifically, assertive acts are intended to inform, directive acts are used to make requests, and expressive acts allow agents to convey emotion.

Existing agent communication languages and methodologies disregard expressives. AOSE methodologies specify messages at a high level and therefore are not granular enough extract the meaning of the messages [2]. On the other hand, agent communication languages specify messages at the appropriate level of detail, but omit expressives. Instead, they have focused on other communicative acts, such as assertives and directives, which can be can be readily incorporated into traditional agent BDI frameworks [15].

3 Koko

Koko's purpose is twofold. It serves as both an affect model container and an agent communication middleware [14]. The affect models that Koko maintains focus on a section of the appraisal theory process (denoted by the dashed box in Fig. 1) in which the models absorb information about the agent's environment and produce an approximation of the agent's affective state. Koko then enables that affective information to be shared among agents via expressive messages. It is important to note that since Koko is designed to model human users, the environment of the agent extends into the physical world. To better report on that environment, Koko supports input from a variety of physical sensors (e.g., GPS devices).

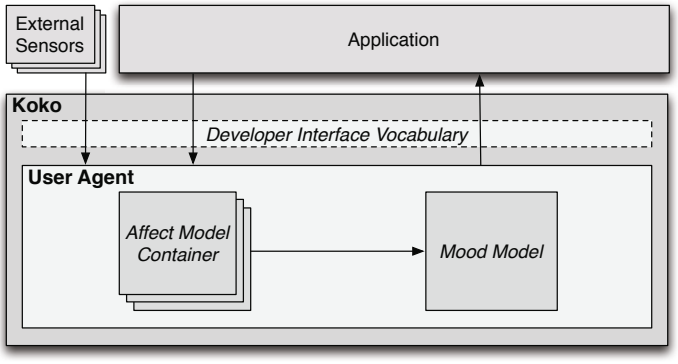


Fig. 2. Koko basic architectural overview

Koko promotes the sharing of affective information at two levels: *cross-user* or interagent communication and *cross-application* or intraagent. For a social (multi-agent) application, Koko enables agents to communicate via expressives. Expressives enable agents to share their current affective state among other agents within Koko (the format of an expressive message is outlined in Section 4). Koko also provides a basis for applications—even those authored by different developers—to share information about a common user. This is simply not possible with current techniques because each application is independent and thereby unaware of other applications being employed by a user.

Fig. 2 shows Koko's basic architecture using arrows to represent data flow. The following sections summarize a few of Koko's key components.

User Agent. Koko hosts an active computational entity or *agent* for each user. In particular, there is one agent per user – the same user may employ multiple Koko-based applications. Each agent has access to global resources such as sensors and messaging but operates autonomously with respect to other agents.

Affect Model Container. This container manages one or more affect models for each user agent. Each application must specify exactly one affect model, which is instantiated for each user of the application. The container then manages that instance for the user agent. As Fig. 3 shows, an application's affect model is specified in terms of the affective

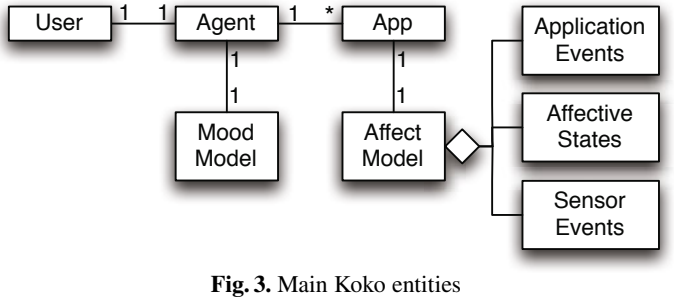


Fig. 3. Main Koko entities

states as well as application and sensor events, which are defined in the application's configuration (described in Section 4) at runtime.

Koko follows CARE's [10] supervised machine learning approach for modeling affect by populating predictive data structures with affective knowledge. This enables Koko to support affect models that depend on an application's domain-specific details, while allowing Koko as a whole to maintain a domain-independent architecture.

For each affect model, the container takes input from the user's physical and application environment and produces an *affect vector*. The resulting *affect vector* contains a set of elements, where each element corresponds to an affective state. The affective state is selected from the emotion ontology that is defined and maintained via the developer interface vocabulary. Using this ontology, each application developer selects the emotions to be modeled for their particular application. For each selected emotion, the vector includes a quantitative measurement of the emotion's intensity. The intensity is a real number ranging from 0 (no emotion) to 10 (extreme emotion).

Mood Model. Following EMA [7], we take an *emotion* as the outcome of one or more specific events and a *mood* as a longer lasting aggregation of the emotions for a specific user. An agent's mood model maintains the user's mood across all applications registered to that user.

For simplicity, Koko's model for mood takes in affect vectors and produces a *mood vector*, which includes an entry for each emotion that Koko is modeling for that user. Each entry represents the aggregate intensity of the emotion from all affect models associated with that user. Consequently, if Koko is modeling more than one application for a given user, the user's mood is a cross-application measurement of the user's emotional state.

Developer Interface Vocabulary. Koko provides a vocabulary through which the application interacts with Koko. The vocabulary consists of two ontologies, one for describing affective states and another for describing the environment. The ontologies are encoded in OWL (Web Ontology Language). If needed, the ontologies are designed to grow to meet the needs of new applications.

The *emotion ontology* describes the structure of an affective state and provides a set of affective states that adhere to that structure. Koko's emotion ontology captures the 24 emotional states proposed by Elliot [3], including states such as *joy*, *hope*, *fear*, and *disappointment*.

The *event ontology* can be conceptualized in two parts: event definitions and events. An event definition is used by applications and sensors to inform Koko of the type of data that they will be sending. The event definition is constructed by selecting terms from the ontology that apply to the application, resulting in a potentially unique subset of the original ontology. Using the definition as a template, an application or sensor generates an event that conforms to the definition. This event then represents the state of the application at a given moment. When the event arrives at the affect model, it is decomposed using the agreed upon event definition.

Koko comes preloaded with an event ontology (partially shown in Fig. 4) that supports common contextual elements such as time, location, and interaction with application objects. Consider an example of a user seeing a snake. To describe this for Koko

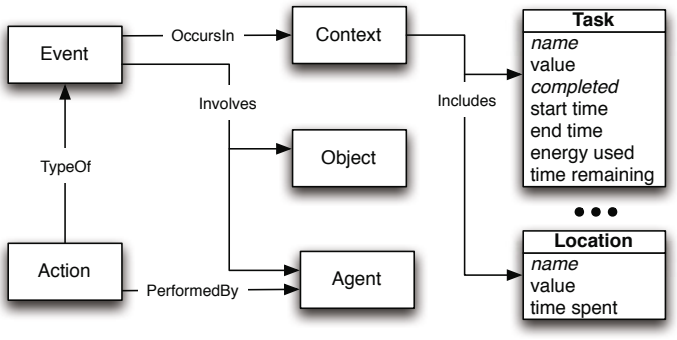


Fig. 4. Event ontology example

you would create an event *seeing*, which involves an object *snake*. The context is often extremely important. For example, the user's emotional response could be quite different depending on whether the location was in a *zoo* or the user's *home*. Therefore, the application developer should identify and describe the appropriate events (including objects) and context (here, the user's location).

Runtime API. The runtime API is the interface through which applications communicate with Koko at runtime. The API can be broken into two discrete units, namely, *event processing* and *querying*. Before we look at each unit individually, it is important to note that the contents of the described events and affect vectors are dependent on the application's initial configuration, which Section 4 discusses.

Application Event Processing. The express purpose of the application-event interface is to provide Koko with information regarding the application's environment. During configuration, a developer defines the application's environment via the event ontology specified in the developer interface. Using the ontology, the developer encodes snapshots of the application's environment. At runtime the snapshots capturing the user's view of the application environment are passed into Koko for processing. Upon receipt, Koko stores each event where it is available for retrieval by the appropriate affect model. This data combined with the additional data provided by external sensors provides the affect model with a complete picture of the user's environment.

Application Queries. Applications query for and retrieve two types of vectors from Koko. The first is an application-specific affect vector and the second is a user-specific mood vector, both of which are modeled using the developer interface's emotion ontology. The difference between the two vectors is that the entries in the affect vector depend upon the set of emotions chosen by the application when it is configured, whereas the mood vector's entries aggregate all emotions modeled for a particular user. As such, a user's mood is relevant across all applications. Suppose a user, Alice, reads an email that makes her angry and the email client's affect model recognizes this. All of Alice's affect enabled applications can benefit from the knowledge that the user is

angry even if they cannot infer that it is from some email. Such mood sharing is natural via Koko because Koko maintains the user's mood and can supply it to any application.

4 Methodology

Now that we have laid the architectural foundation we describe Koko-ASM, a methodology for configuring a social (multiagent) affective application using Koko. Properly configuring an application is key because its inputs and outputs are vital to all of the application interfaces within Koko. In order to perform the configuration, the developer must gather key pieces of information that are required by Koko. Table 1 systematically lists Koko-ASM's steps to create an affective social application. The following documentation concentrates on Steps 1-4, which are of primary interest to AOSE.

Table 1. Koko-ASM: a methodology for creating an affective, social application

Step	Description	Artifacts Produced
1	Define the set of possible roles an agent may assume	Agent Roles
2	Describe the expressives exchanged between roles	Expressive Messages
3	Derive the emotions to be modeled from the expressives	Emotions
4	Describe the set of possible application events	Application Events
5	Select the sensors to be included in the model	Sensor Identifier(s)
6	Select the desired affect model	Model Identifier

Step 1 requires the developer to identify the set of roles an agent may assume in the desired application. Possible roles include TEACHER, STUDENT, PARENT, CHILD, and COWORKER. A single agent can assume multiple roles and a role can be restricted to apply to the agent only if certain criteria are met. For example, the role of COWORKER may only apply if the two agents communicating work for the same company.

Step 2 requires the developer to describe the expressive messages or *expressives* exchanged between various roles [12]. Searle defines expressives as communicative acts that enable a speaker to express his or her attitudes and emotions towards a proposition. Examples include statements like "Congratulations on winning the prize!" where the attitude and emotion is congratulatory and the proposition is winning the prize. Formally, we define the structure of an expressive to match that of a communicative act in general:

$$\langle sender, receiver, type, proposition \rangle \tag{1}$$

The *type* of the expressive refers to the attitude and emotion of the expressive and the *proposition* to its content, including the relevant events. The *sender* and *receiver* are selected from the set of roles defined in Step 1. The developer then formulates the expressives that can be exchanged among agents assuming those roles. The result is the set of all valid expressive messages allowed by the application.

Step 3 requires the developer to select a set of emotions to be modeled from the emotion ontology. The selected emotions are based on the expressives identified in the

previous step. To compute the set of emotions, we evaluate each expressive and select the most relevant emotions from the ontology for that particular expressive. We add the selected emotions to the set of emotions required by the application. This process is repeated for every expressive and the resulting emotion set is the output of this step.

Koko offers support for expressives by providing a well-delineated representation for affect. Koko can thus exploit a natural match between expressives and affect to help designers operationalize the expressives they employ in their applications. Our recommended approach to selecting an emotion is to structure Elliot's set of emotions as a tree (Fig. 5). Each leaf of the tree represents two emotions, one that carries a positive connotation and the other a negative connotation. Given an expressive, you start at the top of the tree and using its *type* and *proposition* you filter down through the appropriate branches until you are left with only the applicable emotions. For example, say that you have a message with a *type* of *excited* and a *proposition* equal to "I won the game." Now using the tree you determine that winning the game is an action the user would have taken and that *excited* has a positive connotation, so the applicable emotion must therefore be *pride*. In general, the sender and receiver would have different interpretations. For example, if the recipient of the above message is the agent who lost the game, then the emotions that are relevant to the recipient would be *admiration* and *reproach* depending on their perception of the winner.

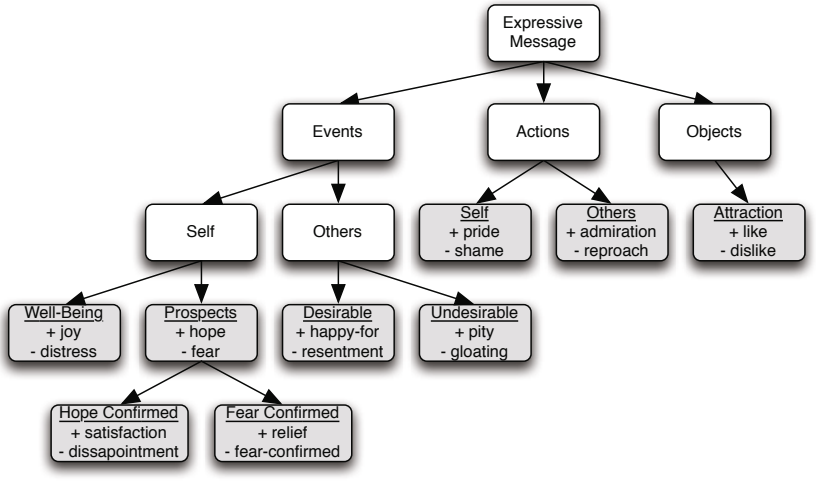


Fig. 5. Expressive message hierarchy

If the *proposition* of the expressive message is composite or even ambiguous as to whether or not the *type* applies to an event, action, or object, then more than one path of the tree may apply. Such is the case when an agent conveys its user's mood via an expressive message. Mood is an aggregation of emotions and therefore does not have a unique causal attribution. For example, an expressive might convey that a user is generally happy or sad without being happy or sad at something. Therefore, we do not select any specific emotion when evaluating a expressive pertaining to mood as the

emotions that comprise the mood are captured when evaluating the other expressives. In other words, mood is not treated directly upon the reception of an expressive.

Step 4 requires the developer to describe the application events using the event ontology. The events described are a combination of the expressives in Step 2 and additional details about the application environment. An expressive is modeled as two application events, one for sending and another for receipt. Each event is modeled as an action (see Fig. 4) in which the sending of a message is described as an action performed by the sender that involves the receiver with the expressive in the context. Similarly, one can envision the receipt of the message as an action performed by the recipient that involves the sender. The decomposition of a message into two events is essential because we cannot make the assumption that the receiving agent will read the message immediately following its receipt and we must accommodate for its autonomy.

The additional details about the application's environment are also modeled using the event ontology. The developer can encode the entire application state using the ontology, but this may not be practical for large applications. Therefore, the developer must select the details about the application's environment that are relevant to the emotions they are attempting to model. For example, the time the user has spent on a current task will most likely effect their emotional status, whereas the time until the application needs to garbage collect its data structures is likely irrelevant. The resulting events are combined with the events derived from the expressive messages to form the set of application events that are needed by Koko.

Step 5 and *Step 6* both have trivial explanations. Koko maintains a listing of both the available sensors and affect models, which are accessible by their unique identifiers. The developer must simply select the appropriate sensor and affect model identifiers.

Based on the artifacts generated by the above methodology we now have sufficient information to configure the Koko middleware. Upon configuration Koko supports affective interactions among agents (using the expressive messages) as well as enables applications to query for the affective state of an agent.

5 Evaluation

Using expressives to communicate an agent's affective state extends traditional AOSE into the world of affective applications. We evaluate Koko-ASM by conducting a case study that steps through the methodology and produces a functional affective social application. The subject of our case study is a social, physical health application with affective capabilities, called booST. To operate, booST requires a mobile phone running Google's Android mobile operating system that is equipped with a GPS sensor.

The purpose of booST is to promote positive physical behavior in young adults by enhancing a social network with affective capabilities and interactive activities. As such, booST utilizes the Google OpenSocial platform [11] to provide support for typical social functions such as maintaining a profile, managing a social circle, and sending and receiving messages. Where booST departs from traditional social applications is in its communication and display of its user's *energy levels* and *emotional status*.

Each user is assigned an *energy level* that is computed using simple heuristics from data retrieved from the GPS sensor on board the phone. Additionally, each user is assigned an *emotional status* generated from the affect vectors retrieved from Koko. The

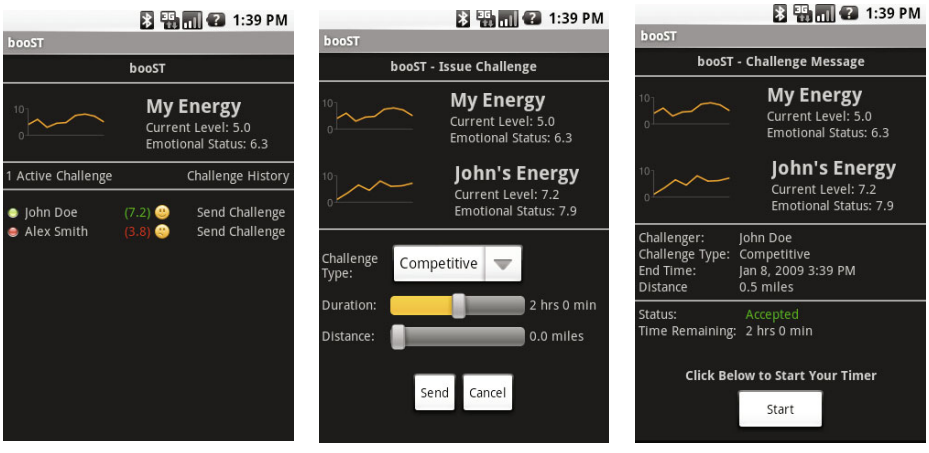


Fig. 6. booST buddy list and activities screenshots

emotional status is represented as a real number ranging from 1 (sad) to 10 (happy). A user's energy level and emotional status are made available to both the user and members of the user's social circle.

To promote positive physical behavior, booST supports interactive physical activities among the members of a user's social circle. The activities are classified as either competitive or cooperative. Both types of activities use heuristics based on the GPS sensor readings to determine the user's progress toward achieving the activities goal. The difference between a competitive activity and a cooperative activity is that in a competitive activity the first user to reach the goal is the winner, whereas in a cooperative activity both parties must reach the goal in order for them to win.

As described in Section 3, Koko hosts an agent for each booST user. A user's agent maintains the affect model that is used to generate the user's emotional status. Further, booST provides the agent with data about its environment, which in this case incorporates the user's social interactions and his or her participation in the booST activities. The user agent processes the data and returns the appropriate emotional status. Further, Koko enables the exchange of affective state between booST agents (representing a social circle of users). This interaction can been seen in Figure 6 in the emoticons next to the name of a buddy. The affective data shared among the members of a social circle provides additional information to the affect model. For instance, if all the members of a user's social circle are sad then their state will have an effect on the user's emotional status.

5.1 Configuring booST

Table 1 outlines Koko-ASM's process for creating an affective, social application. We now demonstrate this process using booST.

Step 1 requires that we identify the set of roles an agent may assume. The booST application involves only two roles, FRIEND and SELF. An agent A assumes the role of

agent B's FRIEND if and only if the users represented by agents A and B are members of each others social circle. The social circle is maintained by booST and can be equated to the friend list in popular social applications such as Facebook and MySpace.

Step 2 requires that we identify and describe all expressives that occur between the two roles. Below is an example of what a few such messages would look like depending on the outcome of a competitive activity within booST. The remaining messages would be defined in a similar fashion.

$$\langle \text{SELF}, \text{FRIEND}, \textit{happy}, \textit{"I won the game"} \rangle \quad (2)$$

$$\langle \text{FRIEND}, \text{SELF}, \textit{sad}, \textit{"I lost the game"} \rangle \quad (3)$$

$$\langle \text{FRIEND}, \text{SELF}, \textit{happy}, \textit{"I ran a good race"} \rangle \quad (4)$$

Step 3 requires that we select a set of emotions to model from the emotion ontology. As Section 3 shows, the ontology is based on Elliot's expansion of the OCC model, which categorizes emotions based on the user's reaction to an action, event, or object. When inspecting each expressive, we find that booST focuses on measuring *happiness* and *sadness* of the user with respect to actions and events. Therefore, we can narrow our selection to only emotions that meet those criteria. As a result, we select four emotions: two are focused on the actions of the user (*pride* and *shame*) and two on the events (*joy* and *distress*). The booST application uses these emotions to compute the user's emotional status by correlating (1) *pride* and *joy* with happiness and (2) *shame* and *distress* with sadness.

Step 4 requires that we describe the application events using the event ontology. Each expressive message yields two events: a sending event and a receiving event. The remaining events provide additional details about the application's environment (Table 2 shows some examples). Since an event in booST is merely an instantiation of the event ontology, the event descriptions are trivial. For example, the "Competitive Exercise Challenge" message can be described as an action that involves another agent. When an event occurs at runtime, the context associated with its occurrence would specify attributes such as the time of day, challenge information, and the user's energy level as calculated by the application.

Step 5 requires that we select the sensors to be included in the model. As we have already noted booST requires a single sensor: a GPS. The first time a sensor is used a plugin must be created and registered with Koko. The GPS plugin converts latitude and longitude vectors into distance covered over a specified time period. This data is then

Table 2. Representative booST events

#	Event Description
1	View my energy level and emotional state
2	View my friend's energy level and emotional state
3	Send or receive "Cooperative Exercise Challenge" message
4	Send or receive "Competitive Exercise Challenge" message
5	Complete or fail a cooperative activity
6	Complete or fail a competitive activity

maintained by Koko and made available for consumption both by the application and the affect model.

Step 6 is trivial as booST employs an affect model that is provided by Koko. In particular, booST employs the model that implements decision trees as its underlying data structure.

Using the artifacts generated by the methodology we now have obtained sufficient information to configure the Koko middleware. Upon configuration, Koko maintains an agent for each booST user. The agent is responsible for modeling the affective state of the user as well as communicating that state to other agents within the user's social circle. With Koko supporting the affective aspects of booST, application developers are free to focus on other aspects of the application. For instance, they may focus on developing the creative aspects of the game, such as how to alter gameplay based on the user's affective state.

6 Discussion

An important contribution of Koko and Koko-ASM is the incorporation of expressive communicative acts. These acts, though well-known in the philosophy of language, are a novelty both in agent-oriented software engineering and in virtual agent systems. The incorporation of expressives enable agents to interpret the difference between an expression of feelings and a statement of fact, thus enabling agents to better model their users and their environment.

Existing Methodologies. Existing AOSE methodologies specify messages at a level that does not describe the contents of the message and therefore are not granular enough to support expressives [2]. Koko-ASM is restricted to applications that are both affective and social, thus its applicability has a much narrower scope than existing methodologies. These distinctions are simply the result of a difference in focus. It is quite possible, given the narrow scope of Koko-ASM, that it could be integrated with broader methodologies in order to leverage their existing processes and tools. For example, many methodologies [2,16] have detailed processes by which they help developers identify all possible messages that are exchanged among agents. Koko-ASM would benefit by integrating those processes, thereby making it easier to identify the expressive messages.

Virtual Agents. Koko and, in particular, Koko-ASM have focused on human-to-human social interactions. This does not inherently limit the methodology only to such interactions. We have begun to explore the application of our methodology on human-to-virtual agent interactions. Using this modified version of Koko-ASM we envision a scenario where the virtual agents will have access to the user's affective state via Koko. Applications that leverage this technique could manipulate a virtual agent's interactions with a user, based on the user's affective state.

Enhanced Social Networking. Human interactions rely upon social intelligence [5]. Social intelligence keys not only on words written or spoken, but also on emotional cues provided by the sender. Koko provides a means to build social applications that can naturally convey such emotional cues, which existing online social networking tools

mostly disregard. For example, an advanced version of booST could use affective data to create an avatar of the sender and have that avatar exhibit emotions consistent with the sender's affective state.

Future Work. Koko and Koko-ASM open up promising areas for future research. As an architecture, it is important that Koko fits in with existing architectures such as game engines. We have described some efforts in a companion paper [14]. Association with other architectures would not only facilitate additional applications but would lead to refinements of the present methodology, which we defer to future research. Further, we are actively working on formalizing the notion of expressive communication with respect to agent communication languages.

References

1. Austin, J.L.: How to Do Things with Words. Oxford University Press, London (1962)
2. Deloach, S.A., Wood, M.F., Sparkman, C.H.: Multiagent Systems Engineering. Journal of Software Engineering and Knowledge Engineering 11(3), 231–258 (2001)
3. Elliott, C.: The Affective Reasoner: A Process Model of Emotions in a Multi-agent System. PhD, Northwestern University (1992)
4. Elliott, C., Rickel, J., Lester, J.C.: Lifelike pedagogical agents and affective computing: An exploratory synthesis. In: Veloso, M.M., Wooldridge, M.J. (eds.) Artificial Intelligence Today. LNCS (LNAI), vol. 1600, pp. 195–211. Springer, Heidelberg (1999)
5. Goleman, D.: Social Intelligence: The New Science of Human Relationships. Bantam Books, New York (2006)
6. Gratch, J., Marsella, S.: Fight the way you train: The role and limits of emotions in training for combat. Brown Journal of World Affairs X(1), 63–76 (Summer/Fall 2003)
7. Gratch, J., Marsella, S.: A domain-independent framework for modeling emotion. Journal of Cognitive Systems Research 5(4), 269–306 (2004)
8. Lazarus, R.S.: Emotion and Adaptation. Oxford University Press, New York (1991)
9. Marsella, S., Johnson, W.L., LaBore, C.: Interactive pedagogical drama. In: International Conference on Autonomous Agents, pp. 301–308 (2000)
10. McQuiggan, S., Lester, J.: Modeling and evaluating empathy in embodied companion agents. International Journal of Human-Computer Studies 65(4) (April 2007)
11. OpenSocial Foundation. Opensocial APIs (2009), http://www.opensocial.org
12. Searle, J.R.: Speech Acts: An Essay in the Philosophy of Language. Cambridge University Press, Cambridge (1970)
13. Smith, C., Lazarus, R.: Emotion and adaptation. In: Pervin, L.A., John, O.P. (eds.) Handbook of Personality: Theory and Research, pp. 609–637. Guilford Press, New York (1990)
14. Sollenberger, D.J., Singh, M.P.: Architecture for Affective Social Games. In: Dignum, F., Bradshaw, J., Silverman, B., van Doesburg, W. (eds.) Agents for Games and Simulations. LNCS (LNAI), vol. 5920, pp. 79–94. Springer, Heidelberg (2009)
15. Vieira, R., Moreira, A., Wooldridge, M., Bordini, R.H.: On the formal semantics of speech-act based communication in an agent-oriented programming language. Journal of Artificial Intelligence Research 29, 221–267 (2007)
16. Zambonelli, F., Jennings, N.R., Wooldridge, M.: Developing Multiagent Systems: The Gaia Methodology. ACM Transactions on Software Engineering and Methodology 12(3), 317–370 (2003)

Automatic Generation of Executable Behavior: A Protocol-Driven Approach

Christian Hahn, Ingo Zinnikus, Stefan Warwas, and Klaus Fischer

German Research Center for Artificial Intelligence (DFKI)
Stuhlsatzenhausweg 3
66123 Saarbrücken
{Christian.Hahn,Ingo.Zinnikus,Stefan.Warwas,Klaus.Fischer}@dfki.de

Abstract. Modern information systems are considered as collection of independent units that interact with each other through the exchange of messages. Especially in the context of multiagent systems, the interaction between agents is of particular importance. Agent interaction protocols (AIPs) are one important mechanism to define agent-based interactions. AIPs play a major role within the platform independent modeling language for multiagent systems (DSML4MAS). In this paper, we demonstrate how to design protocols with DSML4MAS and discuss a model-driven approach to use the protocol description to generate executable code.

1 Introduction

Multiagent systems (MASs) define a powerful distributed computing model, enabling agents to cooperate with each other. Thus, beside, aspects like agents and organizational relationships, the interaction between agents is considered as basic building block of MASs (see [1]). In accordance to [2], an interaction can be viewed as a formalization of a concept of dependence between agents, no matter on whom or how they are dependent. An agent interaction protocol (AIP) as a special case of interactions describes the manner how messages are exchanged.

The importance of interactions in MAS is underlined by the fact that existing methods for designing MAS like Tropos [3], Prometheus [4], Gaia [5], or INGENIAS [6] already included mechanisms to express AIPs. In particular, all of them use some sort of Agent UML diagrams (AUML) [7] for defining agent-based interactions. However, AUML in its current version has some drawbacks that are intended to be resolved by our approach.

Hence, in this paper, we demonstrate how to design protocol-based interactions using a platform independent modeling language for the domain of MASs called DSML4MAS. In [8] an overview of DSML4MAS and a model transformation to the execution platform of JACK Intelligent Agents[1] is given. The model transformation to JACK allows to close the gap between design and implementation

[1] http://www.aosgrp.com/index.html

M.-P. Gleizes and J.J. Gomez-Sanz (Eds.): AOSE 2009, LNCS 6038, pp. 110–124, 2011.

by taking the complete design made with DSML4MAS and transferring it to corresponding concepts of JACK. The idea to apply the principles of model-driven development in agent-oriented software is not new (e.g. [9,10]).

However, especially in a more business-oriented context, partners often first define the manner in which they interact followed by defining the behaviors that actual implement the agreed interactions. Hence, in this paper, we discuss a slightly different approach which is more methodology-like. Instead of checking whether the agent's internal behavior implements the protocol description, we focus on a protocol-driven approach that takes a protocol description as a base and generates a corresponding behavior description that automatically conforms to the agreed AIP. In a second step, the system designer refines the behavior description by adding, for instance, private and critical information. Finally, in a last step, the generated behavior in combination with the remaining design (i.e. agents, organizations, roles, etc.) is transformed to JACK code that in combination with manually written code (if necessary) can be executed.

The remainder of this paper is organized as follows. Section 2 discusses the interaction and behavior view of DSML4MAS in detail. Followed by Section 3, demonstrating how to design AIPs with DSML4MAS. Section 4 discusses our protocol-driven approach by illustrating how to transform AIPs to executable code. Related work is given in Section 5. Finally, a conclusion is drawn.

2 Domain Specific Modeling Language for Multiagent Systems

The domain specific platform independent modeling language for MASs (DSML4MAS) defines a graphical language that allows defining MASs independent of any existing agent-oriented programming language (AOPL). However, model transformations can be applied to automatically generate code in accordance to the AOPLs JACK and JADE. The abstract syntax of DSML4MAS is defined by a metamodel called PIM4AGENTS that consists of several views each focusing on a core building block of MASs.

- *Agent view* defines how to model single autonomous entities, the capabilities they have, and the roles they play within the MAS. Moreover, resources an agent has access to and behaviors used by an agent to solve tasks are specified.
- *Organization view* defines how single autonomous agents are arranged to more complex organizations. Thereby, organizations in PIM4AGENTS can either be an autonomous acting entity like an agent, or simple groups formed to take advantage of the synergies of its members.
- *Role view* covers the abstract representations of functional positions of within organizations or other social relationships like interactions.
- *Interaction view* focuses on the exchange of messages between autonomous entities. Section 2.1 focuses on the interaction view of PIM4AGENTS.

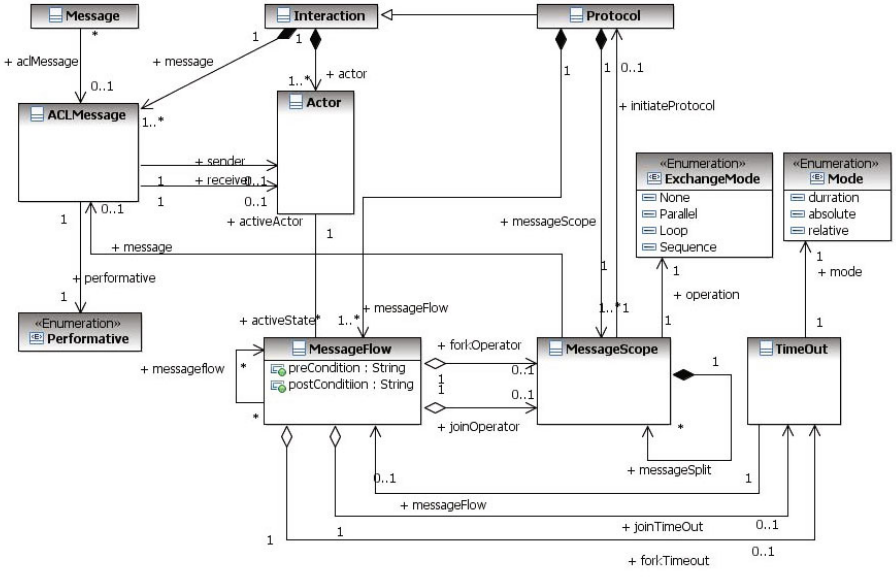


Fig. 1. The partial metamodel reflecting the interaction view of DSML4MAS

- *Behavior view* describes how the internal behavior of intelligent entities can be defined in terms of combining simple actions to more complex control structures or plans. Section 2.2 focuses on the behavior view of PIM4AGENTS.
- *Environment view* contains any kind of entity that is situated in the environment and the resources shared between agents and roles.
- *Multiagent view* contains the core building blocks for describing MASs. In particular, the agents situated in the MAS, the roles they play, and the sorts of interactions.
- *Deployment view* describes the run-time agent instances involved in the system and how these are assigned to roles.

PIM4AGENTS bases on Ecore the meta-metamodel of the Eclipse Modeling Framework[2], a formal semantics of DSML4MAS have been defined in [11]. In the remainder of this section, we lay the foundations for the remaining parts of this paper and discuss the interaction and behavior views in detail, but also mention the links to the remaining views.

2.1 Interaction View

The partial metamodel of the interaction view is depicted in Fig. 1. A *Protocol* is considered as a special form of *Interaction*. Accordingly, the main concepts of a *Protocol* are *Actor*, *ACLMessage*, *MessageFlow*, *MessageScope* and *TimeOut*. In

[2] `http://www.eclipse.org/modeling/emf/`

the deployment view, furthermore, the system designer can specify how *Protocols* are used within *Organizations*. This is done through the concept of *Collaborations* that define which organizational members are bound to which kind of *Actor* as part of an *ActorBinding*.

In DSML4MAS, interaction roles like 'Participant' or 'Initiator' are called *Actors* that bind *AgentInstances* at design time or even at run-time. Actors as a specialization of *Role* can have subactors, where an *AgentInstance* bound to the parent actor must be bound to exactly one subactor. The actor subactor relationship is discussed in more detail in Section 3. Furthermore, *Actors* require and provide certain *Capabilities* and *Resources* defined in the role view of PIM4AGENTS.

Messages are an essential mean for the communication between agents in MASs. In PIM4AGENTS, we distinguish between two sorts of messages, i.e. *Message* and *ACLMessage* which further includes the idea of *Performatives*. *Messages* have a content and may refer to an *Ontology* that can be used by the Agents to interpret its content. A *MessageFlow* defines the states of the AIP in which an *Actor* could be active. The main function of the *MessageFlow* is firstly to send and receive *ACLMessages* which is done through the concept of a *MessageScope* and secondly to specify time constraints (i.e. the latest point in time) in which these *ACLMessages* need to be sent and received through the *TimeOut* concept. A *TimeOut* defines the time constraints for sending and receiving messages and how to continue in case of a *TimeOut* through the *messageFlow* reference.

A *MessageScope* defines the *ACLMessages* and the order how these are sent and received. In particular this is achieved by connecting *ACLMessages* via *ExchangeModes*. Beside *ACLMessages* sent and received, a *MessageScope* may also refer to *Protocols* that are initiated at some specific point in time in the parent *Protocol*. This particular feature allows modeling of nested protocols. The order in which *ACLMessages* are exchanged is defined by a so-called *ExecutionMode* featuring the following alternatives. A *Sequence* defines a sequencing of traces timely ordered. A *Parallel* denotes that several traces are executed concurrently and a *Loop* describes that a particular trace is executed as long as a particular *condition* is satisfied. Finally, *None* specifies that only a single *ACLMessage* is exchanged. A combination of these *ExchangeModes* can easily be achieved by the *MessageScope's messageSplit* reference which allows to nest *ExchangeModes*. Further branching can be defined by specifying transitions between *MessageFlows* using their *messageflow* reference.

2.2 Behavioral View

The behavioral view describes how plans are composed by complex control structures and simple atomic tasks and how information flow between those constructs. The core concepts of the behavioral view are depicted in Fig. 2.

A *Plan* can be considered as a specialization of the abstract *Behavior* to specify an agent's internal processes. An *Agent* can use several *Plans*, each of them contains a set of *Activities* and *Flows* (i.e. *ControlFlow*, *InformationFlow*) that link *Activities* together. The body of a *Plan* is mainly represented by the

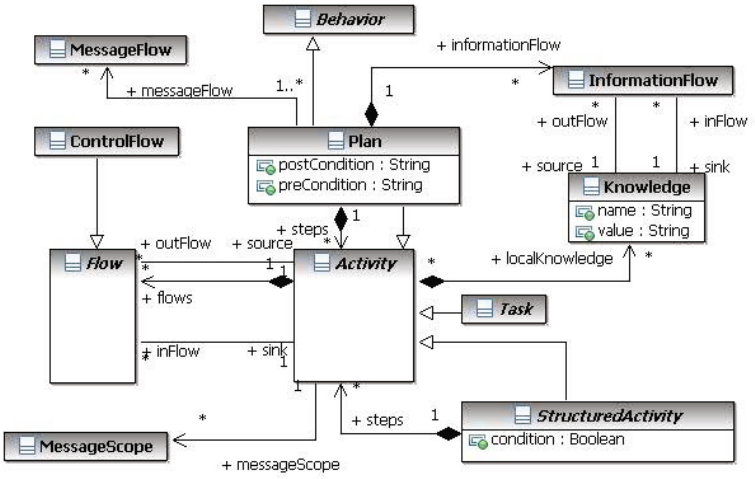


Fig. 2. The partial metamodel reflecting the core behavioral view of DSML4MAS

specializations of an *Activity*. A *StructuredActivity* is an abstract class that introduces more complex control structures into the behavioral view. It inherits from *Activity*, but additionally owns a set of *Activities* and *ControlFlows*.

A *Sequence* as a specialization of a *StructuredActivity* denotes a list of *Activities* to be executed in a sequential manner as defined by contained *ControlFlows* through their *sink* and *source* attributes. A *Split* is an abstract class that defines a point in a *Plan* where a single thread of control splits into multiple threads of control. We distinguish between *Parallel* and *Decision* as specializations of *Split*. A *Parallel* is a point in a *Plan*, where a single thread of control splits into multiple threads of control which are executed in parallel. Thus a *Parallel* allows *Activities* to be executed simultaneously or in any order. How the different threads are synchronized is defined by a *SynchronizationMode*. Feasible options are XOR, AND and NofM (i.e. n of m paths are synchronized). In contrast to a *Parallel*, a *Decision* in PIM4AGENTS is a point in a *Plan* where, based on a *condition*, at least one *Activity* of a number of branching *Activities* must be chosen. A *Decision* can either be executed in an XOR or OR manner. In contrast, a *Loop* is a point in a *Plan*, where a set of *Activities* are executed repeatedly until a certain pre-defined *condition* evaluates to false. It allows looping that is block structured, i.e. patterns allow exactly one entry and exit point. A *ParallelLoop* as a specialization of *Loop* and *Parallel* allows specifying iterations in the form that each trace is executed in parallel.

Like a *StructuredActivity*, a *Task* is an abstract class that inherits from *Activity*. Unlike a *StructuredActivity*, a *Task* mainly focuses on atomic activities and thus does not contain any *Activities* or *ControlFlows*. Sending and receiving messages is covered by the *Send* and *Receive* tasks. Furthermore, the *Wait* activity allows to wait for a particular time or event, the *InternalTask* activity can be used to define code.

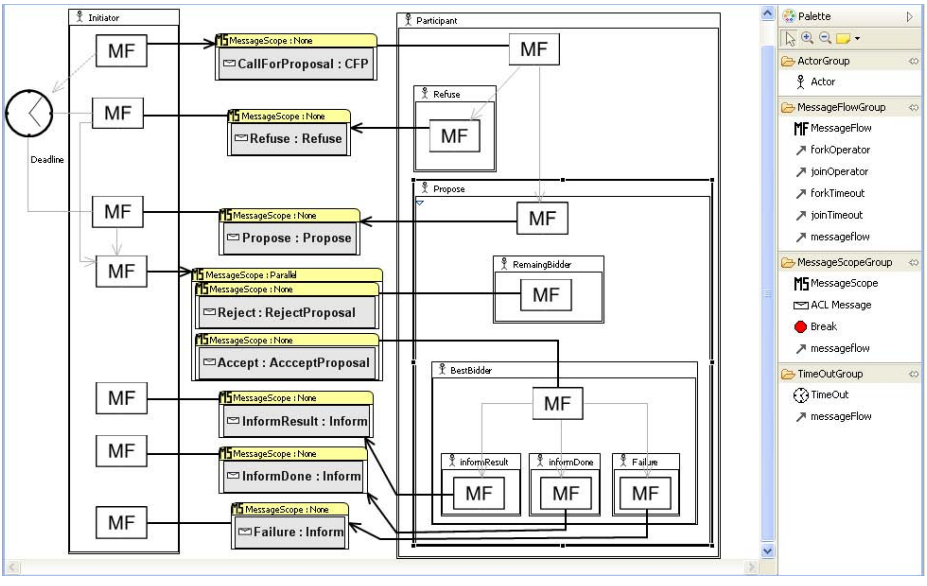


Fig. 3. Contract net protocol modeled using DSML4MAS

3 Example: Contract Net Protocol

In the following, we demonstrate how to model AIPs using DSML4MAS's graphical editor. This editor is based on the abstract syntax (i.e. PIM4AGENTS metamodel) partly introduced in the previous section and uses the Eclipse's Graphical Modeling Framework[3]. Beside the graphical notation that supports modeling of each view in a graphical manner, by clicking on the graphical symbol, the properties of this concept can be further specified and refined in the properties view. A detailed overview on the graphical editor can be found in [12].

For illustrating how to model AIPS, we use the CNP [13] which is depicted in Fig. 3. Therefore, we firstly introduce two actors called *Initiator* and *Participant*. The protocols starts with the first message flow of the *Initiator* that is responsible for sending the *CallForProposal* message of the performative type *cfp*. The *CallForProposal* message specifies the task as well as the conditions that can be specified within the properties view of the graphical editor. When receiving the *CallForProposal*, each agent instance performing the *Participant* decides on the base of free resources whether to *Propose* or *Refuse*. However, how this selection function is defined cannot be expressed in the protocol description, as private information are later manually added to the automatically generated behavior description. To distinguish between the alternatives, two additional actors (i.e. *Propose* and *Refuse*) are defined that are subactors of the *Participant* (i.e. any agent instance performing the *Participant* should either perform the *Propose* or *Refuse* actor). The transitions between the message flow within the

[3] `http://www.eclipse.org/gmf/`

Participant and the message flows of the *Refuse* or *Propose* actors through the *messageFlow* reference underline the change of state for the agent instance performing the *Participant* actor. However, the transitions are only triggered if a certain criterion is met. In the CNP case, the criterion is that the *Initiator* got all replies, independent of its type (i.e. *Refuse* or *Propose*). The *postConditions* of a *MessageFlow* can be defined in the properties view of the graphical editor. The message flows within the *Refuse* and *Propose* actors are then responsible for sending the particular messages (i.e. *Refuse* and *Propose*).

After the deadline expired—defined by the *TimeOut*—or all answers sent by the agent instances performing the *Participant* actor are received, the *Initiator* evaluates the proposals in accordance to a certain selection function, chooses the best bid(s) and finally assigns the actors *BestBidder* and *RemainingBidder* accordingly. Again, the selection function is not part of the protocol, but can be defined later on in the corresponding plan. Both, the *BestBidder* actor as well as the *RemainingBiddder* actor—containing the agent instances that were not selected—are again subactors of the *Propose* actor. The *Initiator* sends an *AcceptProposal* message to the *BestBidder* and a *RejectProposal* message to the *RemainingBidder* in parallel. After completing the work on the assigned task, the *BestBidder* reports its status to the *Initiator* by either sending an *InformResult*, *InformResult* or *Failure* message. For this purpose, we distinguish again between three subactors of *BestBidder*.

4 Model-Driven Methodology to Generate Executable Code

The process of generating executable code based on a protocol description is discussed in this section in more detail. This methodology consists of two phases depicted in Fig. 4. In a first step, a transformation from the protocol description to the internal behaviors of the participating agents is discussed. In a second step, the transformation from the behavioral perspective to the agent-based execution platform JACK is given. Both transformations were implemented utilizing the Atlas Transformation Language[4].

4.1 From Interaction Protocol to Behaviors

Model Transformation. The general approach we take in this model transformation is to initiate one *Plan* for each *Actor* not being referred by any other *Actor* as sub-actor. How this is done concretly is depicted by Mapping Rule 1.

Mapping Rule 1: MessageFlow → Plan

- **name:** name of the *MessageFlow* plus the name of the active *Actor*
- **steps:** collection of *Send* and *Receive* tasks (cf. Mapping Rule 5 and 6), *Parallel*, *Sequence*, and *Loop* activities (cf. Mapping Rules 2-4)
- **flows:** collection of *ControlFlows* combining the generated *Activities*

[4] `http://www.eclipse.org/m2m/atl/`

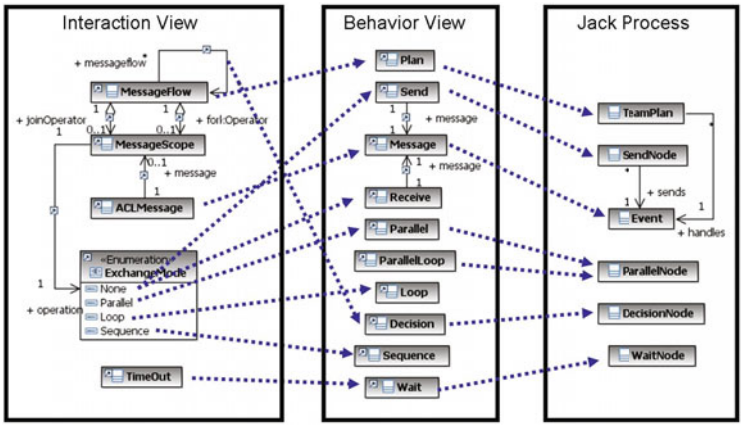


Fig. 4. Conceptual model transformations between (i) the interaction view and behavioral view (left hand side) and (ii) the behavioral view and JackMM plan aspect (right hand side)

A *MessageFlow* is considered as a state an *Actor* is within an *Interaction*, where mainly to actions are feasible, namely sending and receiving *ACLMessages*. Even if Mapping Rule 1 only transforms the first *MessageFlow* of any active *Actor* directly, any subsequent *MessageFlow* of the same *Actor* and relevant information with respect to (i) the *MessageScopes* used by the subsequent *MessageFlows* and (ii) potential *Timeouts* is collected in order to form the body of the *Plan* by applying the corresponding mapping rules.

The *Plans* generated for one *Actor* are collected and included in the *Actors* provided *Capabilities*. Any *Agent* which may possibly be bound to this *Actor* through its *DomainRole* will have access to this *Behavior* to make use of. Beside instantiating a *Plan*, a *MessageFlow* is also the source for generating *Decisions*. However, only *MessageFlows* are considered that have a non-empty *messageflow* reference. This means that in these particular *MessageFlows*, the *Actor* is split into two or more subactors. This split may base on a condition that will be exactly specified within the *Decision* introduced in the generated *Plan*.

Within the body of a *Plan*, the order of sending and receiving *Messages* is deduced from the order in which the corresponding *MessageScopes* are arranged and to which *ExchangeMode* (i.e. *None, Parallel, Loop* or *Sequence*) they refer. This means in particular that a *Parallel* operation is mapped to a *Parallel* activity, a *Sequence* operation is mapped to a *Sequence* activity, and finally a *Loop* operation is mapped to a *Loop* activity as defined by the following three mapping rules.

Mapping Rule 2: ExchangeMode:Parallel → Parallel

- **name:** name of the *ExchangeMode's MessageScope*
- **flows:** collection of *ControlFlows* combining the generated *Activities*

- **steps:** collection of *MessageScopes* part of the *MessageScope's messageSplit* reference addressing this *ExchangeMode* through the *operation* attribute

For any kind of *MessageScope* part of the particular *MessageScope* of type *ExecutionMode:Parallel* a unique trace of the particular *Parallel* activity is defined which may consists of any other combination of complex (e.g. *Sequence*) or simple (e.g. *Send*, *Receive*) control structure. How to transform *EchangeMode:Sequence* is depicted in Mapping Rule 3.

Mapping Rule 3: ExchangeMode:Sequence → Sequence

- **name:** name of the *ExchangeMode's MessageScope*
- **flows:** collection of *ControlFlows* combining the generated *Activities*
- **steps:** collection of *MessageScopes* part of the *MessageScope's messageSplit* reference addressing this *ExchangeMode* through the *operation* attribute

In contrast to *ExchangeMode:Parallel*, in the case of a *Sequence*, a unique trace is specified. The order in which the *Activities* are arranged is deduced from the order (from top to bottom) of the contained *MessageScope*.

Mapping Rule 4: ExchangeMode:Loop → Loop

- **name:** name of the *ExchangeMode's MessageScope*
- **flows:** collection of *ControlFlows* combining the generated *Activities*
- **steps:** collection of *MessageScopes* part of the *MessageScope's messageSplit* reference addressing this *ExchangeMode* through the *operation* attribute.

Like in the manner of a *Sequence*, for any *ExchangeMode:Loop*, a unique trace is initiated. The order is again deduced from the ordering of contained *MessageScopes*.

In contrast to the different types like *Parallel*, *Loop*, or *Sequence*, a *ExchangeMode:None* refers to exactly one *ACLMessage*. As any kind of message needs to be sent but also received, the particular *MessageScope* is either mapped to a *Receive* or *Send* task, depending on whether the corresponding *MessageFlow* sends or receives the *ACLMessage*.

Mapping Rule 5: ExchangeMode:None → Send

- **name:** name of the *ExchangeMode's MessageScope*
- **message:** reference to the transformed *ACLMessage* (cf. Mapping Rule 7) that is referred by the *ExchangeMode's MessageScope*

Mapping Rule 6: ExchangeMode:None → Receive

- **name:** name of the *ExchangeMode's MessageScope*
- **message:** reference to the transformed *ACLMessage* (cf. Mapping Rule 7) that is referred by the *ExchangeMode's MessageScope*

The *Send* and *Receive* activities generated by the Mapping Rules 5 and 6 refer to a *Message* that is generated by applying Mapping Rule 7. As *ACLMessages* are sent and received to/from multiple *Actors* that are possibly performed by multiple *AgentInstances*, both, *Send* and *Receive*, have to be included in a parallel statement (i.e. *ParallelLoop*) that iterates over all *AgentInstances* currently performing this *Actor*. This allows keeping a *Plan* as generic as possible, as the information how many *AgentInstances* are finally playing the particular *Actors* does not need to be known at design time. The next rule deals with the mapping between *ACLMessages* and *Messages*.

Mapping Rule 7: ACLMessage → Message

- **name:** name of the *ACLMessage*
- **aclMessage:** reference to the target *ACLMessage*

Mapping Rule 7 mainly defines the type of the *Message* that is referred by *Send* and *Receive* activities. However, as a *Protocol* does not provide any information concerning the message's content, the *Message's* content slot needs to be filled manually. The same holds for the *sender* and *receiver* slot, as the *AgentInstance* sending and receiving the particular *Message* is in the most cases not available during design-time, but most properly appointed at run time.

Finally, the last basic rule deals with the mapping of *TimeOuts*. Therefore, each *TimeOut* is mapped in the manner that a *Wait* activity is introduced and integrated into a *Parallel* activity consisting of two paths. One path includes the *Wait* activity, the other one includes the *Activities* responsible for sending and receiving messages within the given time frame.

Mapping Rule 8: TimeOut → Wait

- **name:** name of the *TimeOut*
- **timeout:** reference to the target *TimeOut*

Mapping Rule 8 generates a *Wait* activity that refers to a single *TimeOut* to specify the time the particular *Agent* has to wait.

Generated Behaviors. Fig. 5 depicts the generated initiator's plan for collecting the initial responses from the participant's side and sending of accept and reject messages. It bases on the message flows responsible for receiving the *Refuse* and *Propose* messages and sending the *Reject* and *Accept* messages. The plan mainly consists of three phases. In the first phase, the *CollectResponses* parallel is triggered that has two paths, one responsible for waiting until the particular *TimeOut* is raised and one for collecting the responses. The execution mode of this parallel statement is XOR, meaning that the statement can be left after all messages were received or a certain time—which is defined by the *TimeOut* in the *Protocol*—has been waited. The execution mode of a parallel statement can be selected within the properties view of the graphical editor. This is also the case for *Messages* referred to by *Send/Receive* and *TimeOuts* referred to by *Wait*.

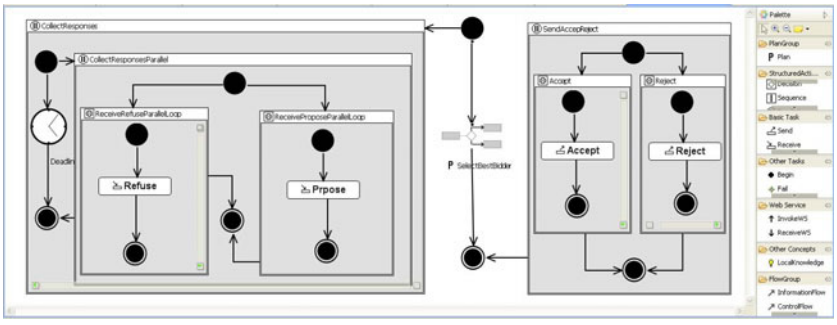


Fig. 5. The generated *SendAcceptReject* plan

The messages are collected inside a *Parallel* called *CollectResponsesParallel*. For each entity bound to the *Participant* actor, either the *ParallelLoop ReceiveProposeParallelLoop* or the *ParallelLoop ReceiveRefuseParallelLoop* is executed.

In the second phase, the *SelectBestBidder* plan is triggered which needs to be added manually as the information according to which criteria the best bidders are selected is not part of the protocol description. After an allocation has been evaluated, in the last phase, the agent instances are assigned to the corresponding actors *BestBidder* and *RemainingBidder* and informed accordingly. This is done in the *SendAcceptReject* parallel, where the *Reject* and *Accept* messages are sent to the *RemainingBidder* and *BestBidder* concurrently. Again, the send tasks are integrated into a parallel loop activity specifying that for each agent instance bound to one of the actors either the message *Accept* or *Reject* is sent.

4.2 From Behaviors to JACK

JACK is a process-centric agent-based programming language that bases on principles of the belief-desire-intention theory [14]. JACK mainly focuses on the internal perspective of an agent defined by so-called plans, an interaction protocol perspective between agents is not provided. In [15], we defined a metamodel for JACK called JackMM that distinguishes between the team view and the process view. The right hand side of Fig. 4 depicts some concepts of the process view briefly discussed in the following. A *TeamPlan* in JackMM is a special *Plan* that can be used within a *Team* to organize its members. Normally, a *TeamPlan* handles exactly one and may send several *Events* which is pretty similar to *Messages* in DSML4MAS (at least those *Events* that are of the type message event). Several activities like *SendNode*, *ParallelNode*, *DecisionNode* and *Flows* are available within a *TeamPlan* to define how to achieve a certain goal. A more complete overview on JackMM-related concepts can be found in [15].

The conceptual transformation from the behavioral metamodel of DSML4MAS to the JACK process metamodel is illustrated on the right hand side of Fig. 4. For each *Plan* generated by applying the interaction to behavior transformation, a *TeamPlan* is generated. The body of this *TeamPlan* is generated in an one-to-one

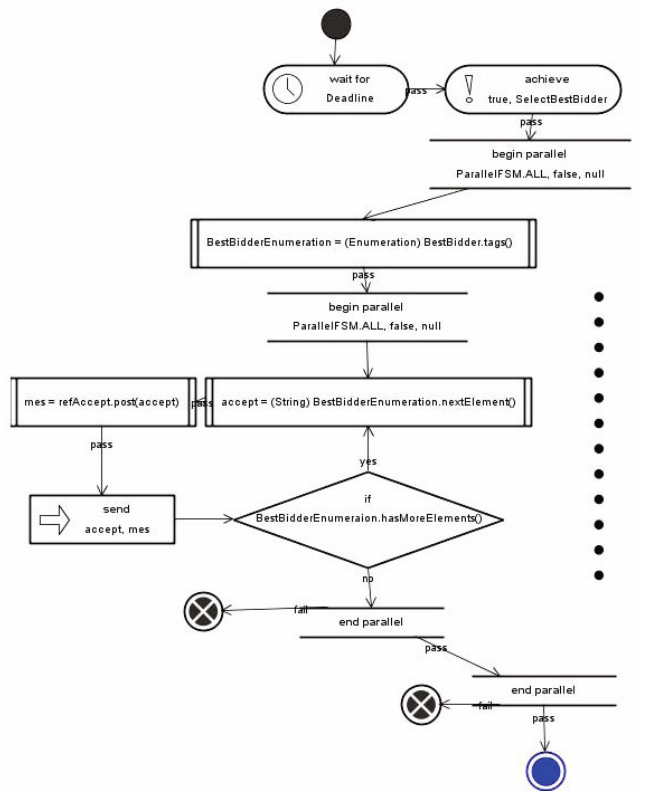


Fig. 6. The generated *SendAcceptReject* team plan in JACK (partial)

manner at least for those activities in PIM4AGENTS that were directly supported by JACK. An example is the *Send* activity which is transformed to a *SendNode* in JACK. The *Message* a *Send* refers to is directly transformed to the corresponding *Event*. As a *TeamPlan* automatically handles an *Event*, we do not need to directly transfer *Receive* activities of DSML4MAS to related concepts in JACK. However, as *Plans* in DSML4MAS base on *MessageFlows* which clearly describe which *Messages* are received and sent, we can easily generate the *TeamPlan*-specific structure. Concepts like *Parallel*, *Decision* and *Wait* can also be mapped in an one-to-one fashion as illustrated in Fig. 4. In the contrast to concepts like *Loop*, *ParallelLoop* and *Sequence* which are not directly supported by JACK. In the case of *Sequences* this can easily be compensated by connecting the predecessor of a *Sequence* with the first activity within the *Sequence* and the last activity of a *Sequence* with its successor. For the concepts of *Loop* and *ParallelLoop*, we define templates consisting of several activities of the process view (e.g. *Decision* concept used for looping purpose) and filled with the activities contained by the source concepts (i.e. *Parallel* and *ParallelLoop*).

Fig. 6 depicts the generated *SendAcceptReject* team plan. Due to space restrictions, we mainly focus on the part responsible for sending the *Accept* event.

Though, except the names of the variables and the event sent, the part of sending the *Reject* event is rather similar. As depicted in Fig. 4 and mentioned earlier, for each *Plan* in DSML4MAS, we generate a corresponding *TeamPlan*. Hence, the corresponding team plans *ReceivePropose* and *ReceiveRefuse* are not part of *SendAcceptReject*. Those are automatically triggered whenever a *Refuse* and *Propose* event is sent by the any participant. Both plans are responsible for updating the belief set of the particular agent with the information whether the replying agent instance refuses or proses.

After the deadline expires or all answers were received, the *SelectBestBidder* team plan is invoked using the concept *achieve*. Internally, the *achieve* statement posts an internal event which triggers the *SelectBestBidder* plan and waits until the best bidder and remaining bidders were selected. Afterwards, the evaluation is reported to the proposed agent instances. Like in the corresponding plan of DSML4MAS illustrated in Fig. 5, this is done using the parallel statement of JACK. For each, the best bidders and the remaining bidders, a path is instantiated responsible for sending the particular event. The process of sending is again done in a parallel statement. However, in this case, the parallel works as a parallel iteration, as for each agent instance of the best bidders (i.e. `accept`) an event `mes` is sent. The loop terminates if every instance is addressed which is ensured by the code `BestBidderEnumeraion.hasMoreElements()` part of the decision.

5 Related Work

In the MAS community, Agent UML (AUML) is the most prominent modeling language for specifying AIPs. AUML is an extension of the Unified Modeling Language (UML) to overcome the limitations of UML with respect to MAS development. In particular, AUML specifies AIPs by providing mechanisms to define agent roles, agent lifelines (interaction threads, which can split into several lifelines and merge at some subsequent points using connectors like AND, OR or XOR), nested and interleaved protocols (patterns of interaction that can be reused with guards and constraints), and extended semantics for UML messages (for instance, to indicate the associated communicative act, and whether messages are synchronous or not). However, AUML does not allow to express more specialized subactors. For this purpose, in [16], Haugen suggested two improvements to the UML 2.0 sequence diagram notation in order to define multicast messages and combined-fragment iterators over subsets. This approach share several commonalities with our approach. However, the suggested improvements are not part of the recent version of AUML. Even if AUML can be considered as de facto standard for modeling AIPs, tool supported is very limited. In [17], a textual notation and graphical tool have been presented. In [18], an approach is presented that automatically interprets AUML AIPs. However, the resulting tool called Paul (Plug-in for Agent UML Linking) only supports parts of AUML as only the alternative operator is implemented. Furthermore, the code generation is limited to two agent lifelines and it is pretty unclear if multicast messages are supported.

6 Conclusion

This paper discusses an approach to describe agent interactions in a protocol-based manner. For this purpose, we discussed the interaction and behavioral parts of our domain specific modeling language for MAS (DSML4MAS) and illustrated how to use DSML4MAS to design the contract net protocol. The approach we are taking in the interaction view is intended to express that there are different actors relative to the initiator described in the protocol. Rather than defining one actor that is fully general, we now describe one actor for each distinct situation. Hence, protocols like the CNP can easily be described using the interaction view of DSML4MAS as one-to-many interactions as well as the differentiation between subactors of the same actor is naturally supported. This is a very interesting result as AUML lacks this kind of expressiveness.

Beside demonstrating how to model AIPs using DSML4MAS we furthermore discussed how to transform agent interaction protocols designed to executable behaviors by applying principles of model-driven development. Therefore, we firstly transformed the interaction description to a process-centric model in DSML4MAS. Therefore, we defined a mapping between concepts of the interaction and the behavioral metamodel. This was mainly done in an one-to-one manner, however, private information needs to be integrated in the agents' internal behavior to make each agent's behavior complete regarding execution. In a next and last step, the DSML4MAS model including the generated behavioral model is mapped to the agent-based programming language JACK which finally allows to execute the protocol description.

References

1. Jennings, N.R.: An agent-based approach for building complex software systems. Communications of the ACM 44(4), 35–41 (2001)
2. Malone, T., Crowston, K.: The interdisciplinary study of coordination. ACM Computing Surveys 26(1), 87–119 (1994)
3. Susi, A., Perini, A., Giorgini, P., Mylopoulos, J.: The tropos metamodel and its use. Informatica 29(4), 401–408 (2005)
4. Padgham, L., Thangarajah, J., Winikoff, M.: AUML protocols and code generation in the prometheus design tool. In: AAMAS 2007: Proceedings of the 6th International Joint Conference on Autonomous Agents and Multiagent Systems, pp. 1–2. ACM, New York (2007)
5. Cernuzzi, L., Zambonelli, F.: Experiencing AUML in the GAIA methodology. In: Proceedings of the 6th International Conference on Enterprise Information Systems, Porto, Portugal, April 14-17, pp. 283–288 (2004)
6. Pavón, J., Gómez-Sanz, J.J.: Agent oriented software engineering with INGENIAS. In: Mařík, V., Müller, J.P., Pěchouček, M. (eds.) CEEMAS 2003. LNCS (LNAI), vol. 2691, p. 394. Springer, Heidelberg (2003)
7. Bauer, B., Odell, J.: UML 2.0 and agents: How to build agent-based systems with the new UML standard. Journal of Engineering Applications of AI 18(2), 141–157 (2002)

8. Hahn, C.: A domain specific modeling language for multiagent systems. In: Padgham, L., Parkes, C.P., Mueller, J., Parsons, S. (eds.) Proceedings of 7th International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS 2008), pp. 233–240 (2008)
9. Rougemaille, S., Arcangeli, J.-P., Gleizes, M.-P., Migeon, F.: ADELFE Design, AMAS-ML in Action. In: Artikis, A., Picard, G., Vercouter, L. (eds.) ESAW 2008. LNCS, vol. 5485, pp. 105–120. Springer, Heidelberg (2009)
10. Jayatilleke, G., Thangarajah, J., Padgham, L., Winikoff, M.: Component agent framework for domain-experts (cafne) toolkit. In: Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2006), pp. 1465–1466. ACM, New York (2006)
11. Hahn, C., Fischer, K.: The formal semantics of the domain specific modeling language for multiagent systems. In: Luck, M., Gomez-Sanz, J.J. (eds.) AOSE 2008. LNCS, vol. 5386, pp. 145–158. Springer, Heidelberg (2009)
12. Warwas, S., Hahn, C.: The contrete syntax of the platform independent modeling language for multiagent systems. In: Proceedings of the Agent-based Technologies and Applications for Enterprise InterOPerability, ATOP 2008 (2008)
13. Smith, R.G.: The contract net protocol: high-level communication and control in a distributed problem solver, pp. 357–366 (1988)
14. Rao, A.S., Georgeff, M.P.: Modeling agents within a BDI-architecture. In: Fikes, R., Sandewall, E. (eds.) Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning (KR 1991), Cambridge, Mass., pp. 473–484. Morgan Kaufmann, San Francisco (1991)
15. Fischer, K., Hahn, C., Madrigal-Mora, C.: Agent-oriented software engineering: a model-driven approach. International Journal on Agent-Oriented Software Engineering 1(3/4), 334–369 (2007)
16. Haugen, O.: Challanges to UML 2 to describe FIPA Agent Protocol. In: Proceedings of Agent-based Technologies and applications for enterprise interOPerability (ATOP 2008). Workshop held at the Seventh International Joint Conference on Autonomous Agents & Multiagent Systems, pp. 37–46 (2008)
17. Winikoff, M.: Towards making Agent UML practical: A textual notation and a tool. In: QSIC 2005: Proceedings of the Fifth International Conference on Quality Software, Washington, DC, USA, pp. 401–412. IEEE Computer Society, Los Alamitos (2005)
18. Ehrler, L., Cranefield, S.: Executing agent uml diagrams. In: AAMAS 2004: Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems, pp. 906–913. IEEE Computer Society, Los Alamitos (2004)

On the Development of Multi-agent Systems Product Lines: A Domain Engineering Process

Ingrid Nunes[1], Carlos J.P. de Lucena[1], Uirá Kulesza[2], and Camila Nunes[1]

[1] PUC-Rio, Computer Science Department, LES - Rio de Janeiro, Brazil
{ionunes,lucena,cnunes}@inf.puc-rio.br
[2] Federal University of Rio Grande do Norte (UFRN) - Natal, Brazil
uira@dimap.ufrn.br

Abstract. Multi-agent System Product Lines (MAS-PLs) are the integration of two promising technologies: Multi-agent Systems (MASs), which provides a powerful abstraction to model features with autonomous and pro-active behavior, and Software Product Lines (SPLs), whose aim is to reduce both time-to-market and costs in the development of system families by the exploitation of commonalities among family members. This paper presents a domain engineering process for developing MAS-PLs. It defines activities and work products, whose purposes include allowing agent variability and providing agent features traceability, both not addressed by current SPL and MAS approaches.

Keywords: Multi-agent Systems, Software Product Lines, Domain Engineering, Software Process.

1 Introduction

Complex modern software systems tend to be situated, open, autonomous and highly interactive [1]. Agent-oriented Software Engineering (AOSE) has emerged as new paradigm that addresses the development of complex and distributed systems based on their decomposition into autonomous and pro-active agents, which together compose a Multi-agent System (MAS). However, MAS methodologies have not addressed so far the need of developing large scale customized systems and little effort has been done to take advantage of software reuse techniques. Software Product Lines (SPLs) manage to promote reduced time-to-market, lower development costs and higher quality to the development of applications that share common and variable features. A feature is a system property that is relevant to some stakeholder and is used to capture commonalities or discriminate among products in a SPL. Based on the exploitation of application commonalities and large-scale reuse, these applications are derived in a systematic way and are customized to specific user needs. In order to fulfil the increasing demand of large-scale and customized MASs, Multi-agent System Product Lines (MAS-PLs) have emerged to integrate these two promising trends of software engineering. The main goal of MAS-PLs is to incorporate their respective benefits and to help the industrial exploitation of agent technology.

M.-P. Gleizes and J.J. Gomez-Sanz (Eds.): AOSE 2009, LNCS 6038, pp. 125–139, 2011.

In this context, this paper presents a domain engineering process for developing MAS-PLs. The two main issues in the MAS-PL development that we aim at addressing and have not been addressed by current approaches are: (i) *documenting agent variability* – explicit variability documentation is essential in SPLs [2]. Nevertheless, SPL approaches do not cover variability documentation in agent models; they focus on specific models, e.g. object-oriented [3] and component-oriented [4]; and (ii) *tracing agent features* – feature traceability allows to specify the configuration knowledge between problem and solution space thus enabling the selection of appropriate artifacts of a SPL in the product derivation process. In defining our process we introduce new and modified models as well as leverage some activities and notations that consist of parts of methods of existing SPL and MAS approaches [3,5,6].

Besides providing customized applications derived in a systematic way, the scenario we are currently exploring is the incorporation of autonomous and pro-active behavior to existing web systems. This is becoming common practice for several web-based systems, for instance recommending products in online stores and displaying personalized advertisements in search engines according to previous searches. We have distinguished these features that provide autonomous and pro-active behavior by naming them as agent features. Consequently, we aim at explicitly separating agent and non-agent features, mainly due to two reasons: (i) agent abstraction provides some particular characteristics, such as autonomy and pro-activeness. Features that do not require them can be modeled and implemented using other technologies (e.g. object-oriented) taking benefit from frameworks and approaches already proposed; and (ii) we aim at supporting the evolution of existing applications and SPLs that have been developed using other existing technologies by the incorporation of agent features. Given that we are adopting a feature-oriented development approach, features are modeled independently of each other. Therefore, it is possible to modeling agent and non-agent features in different ways.

The paper is structured as follows. Related work is presented in Section 2. Section 3 presents the proposed domain engineering process, first giving an overview of it, and later detailing each of its phases. Section 4 concludes this paper and points out directions for future work.

2 Existing MAS-PL Approaches

Several approaches have been published to address problems and challenges of both SPL and MAS engineering [3,5,6,7]. Even though many MAS methodologies have been proposed, most of them do not take into account the adoption of extensive reuse practices that can bring an increased productivity and quality to the software development [8]. They do not consider variability on agent models and do not take into account feature modularization and traceability. Despite the fact that SPL approaches provide useful notations to model agent features, none of them completely covers all their properties and concepts [9]. They do not provide models to design agent concepts and map them to features.

Only few attempts have explored the integration synergy of MASs and SPLs. Pena et al. [10] propose an approach based on MaCMAS methodology, which consists of using goal-oriented requirement documents, role models, and traceability diagrams in order to build a first model of the system. A principle of SPLs is to design and implement features as modularized as possible in order to allow an effective application engineering. However, their approach proposes that variabilities are analyzed after modeling the MAS, and this can lead to undesired situations, such as, the high coupling between mandatory and optional features and inadequate modularization of agent features.

Dehlinger & Lutz [11] have proposed an extensible agent-oriented requirements specification template for distributed systems that supports safe reuse. Their proposal adopts a SPL approach to promote reuse in MASs, which was developed using the Gaia methodology. Although this approach provides a template to capture agent variability, it covers only the requirements engineering phase, and therefore it does not offer a complete solution to address the modeling of agent features in the domain design and implementation.

3 A Domain Engineering Process for MAS-PL

SPL engineering [7,4] aims at improving the development of system families by exploiting common features of applications. A SPL is defined as [7] "a set of software intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way." SPL engineering allows a systematic derivation of products of the same family from a flexible architecture that supports variability. It is typically composed of two key processes: domain engineering and application engineering. Domain engineering is the process of SPL engineering in which SPL commonalities and variabilities are identified, defined and realized. During the application engineering, applications of the SPL are built by reusing domain artifacts and exploiting the SPL variability. The process of deriving products based on reusable domain artifacts is called product derivation process. This section first introduces our domain engineering process and describes its key characteristics (Section 3.1). Later, it presents and analyzes approaches that led to some activities and notations incorporated to our process. (Section 3.2). Finally, it details each phase of our domain engineering process (Sections 3.3, 3.4 and 3.5).

We illustrate our process phases with our ExpertCommittee (EC) case study [12]. It is a MAS-PL of conference management systems, whose aim is to manage paper submission and reviewing processes from conferences. The multi-agent version of this kind of system was first proposed in [13] and since has been widely used to the elaboration and application of MAS methodologies. We assume the readers of this paper are mostly knowledgeable about the domain, but a complete description about this case study can be seen in [12]. The EC MAS-PL was developed in a stepwise fashion. The first version comprises the MAS-PL core built with object-oriented technology, providing mandatory features. Later, new

features were added to this core: reviewer role and functionalities related to it; automatic suggestion of conferences to authors; message notifications to users through email or SMS (alternative feature); and automatic assignment of papers to committee members. Most of these new features have the goal of automating tasks previously done by users. Due to the pro-active and autonomous nature of these features, they were developed using agent technology. Products may be derived from the MAS-PL by the selection of optional and alternative features.

Due to space restrictions, we focus on describing activities purposes and their main output work products, suppressing some details such as tasks of each activity and roles. Additional details and work products of the EC case study can be found in [14].

3.1 Process Overview

Our process is structured according to the SPEM [15], which provides a common syntax and modeling structure to construct software process models. It is based on three main levels: (i) phases – significant periods in a process; (ii) activities – general units of work; and (iii) tasks – define work being performed by roles and are associated with input and output work products. Figure 1 summarizes our process. Following typical domain engineering processes, our approach encompasses three phases: (i) Domain Analysis: the main concepts and activities in a domain are identified and modeled using adequate modeling techniques. Common and variable parts of a system family are identified; (ii) Domain Design: its purpose is to develop a common system family architecture and production plan for the SPL; and (iii) Domain Realization: it involves implementing the architecture, components, and the production plan using appropriate technologies. The qualifier "domain" emphasizes the multisystem scope of these phases. Figures 2 and 3 present most of the work products used in the Domain Analysis and Domain Design phases, respectively. showing relationships among models.

Our process aggregates activities and notations that are specific to model agents and their variabilities to address MAS-PLs. Notations and guidelines that have been adopted alongside all our process are: (i) use of $\ll$*kernel*$\gg$, $\ll$*optional*$\gg$ and $\ll$*alternative*$\gg$ stereotypes to indicate variability in different model elements (e.g. use cases, classes and agents) of several models; (ii) separated modeling of features, to stress the fact that diagrams are split accordingly; (iii) specific models to provide features traceability along all the process. Most activities have a specific task for generating the traceability model; and (iv) use of colors to structure models in terms of features. A different color is attributed for each feature and this color is used in all model elements related to the feature. This is a redundant information used to provide a better visualization of features traceability, even though it is already provided by dependency models.

3.2 Method Fragments Incorporated to Our Process

Even though SPL and MAS approaches present deficiencies to develop MAS-PLs, they provide useful notations and activities that can be integrated to model

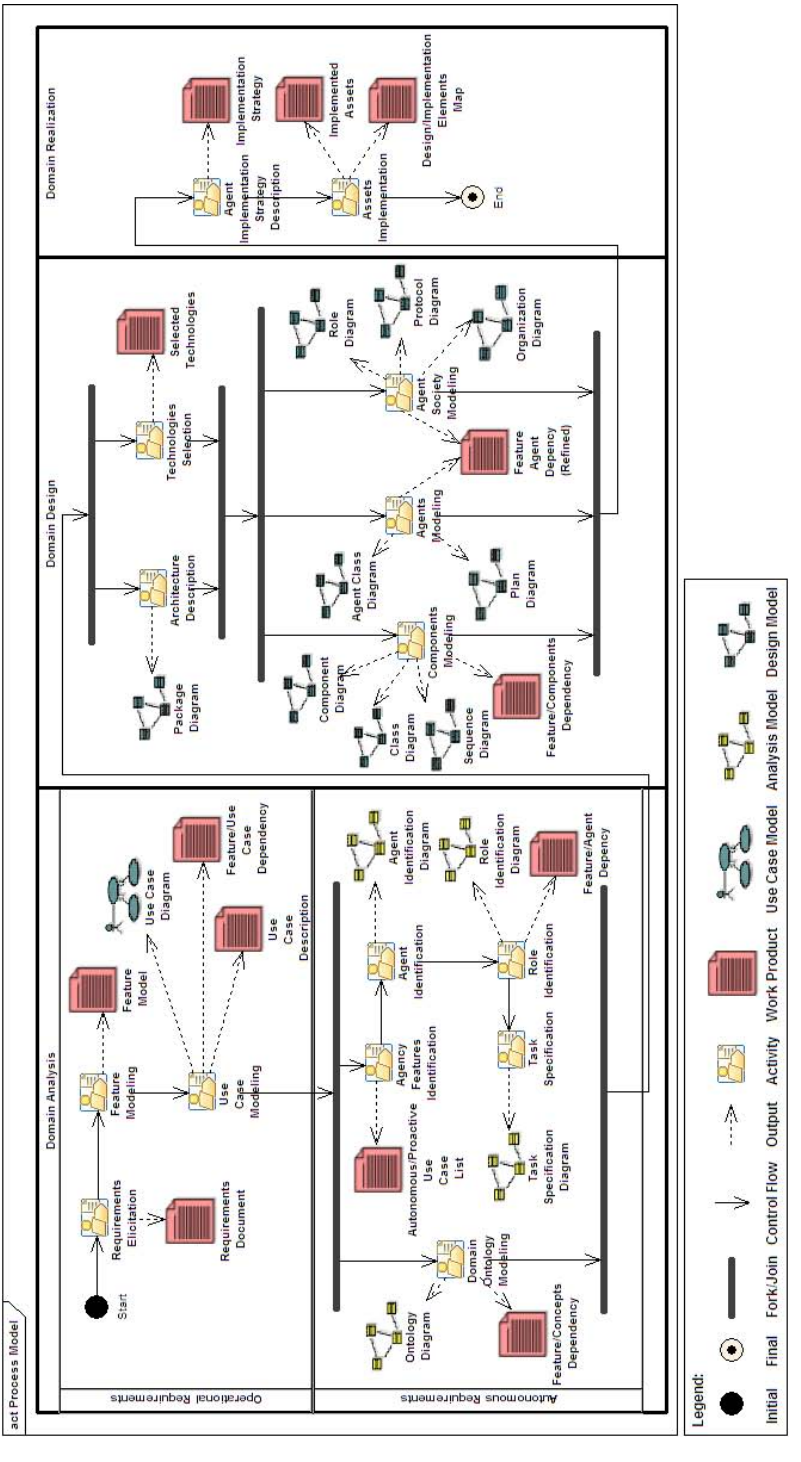


Fig. 1. The Domain Engineering Process

MAS-PLs. Consequently, instead of proposing an approach from scratch, we have incorporated fragments of existing approaches into our process.

The PLUS method [3] provides a set of concepts and techniques to extend UML-based design methods and processes for single systems to handle SPLs. Basically the reasons for adopting PLUS are [9]: (i) it explicitly models the commonality and variability in a SPL, mainly through the use of UML stereotypes; (ii) it uses feature modeling to address variability in the domain analysis, as it is commonly done in SPL approaches. On the other hand, other SPL approaches have the following drawbacks: they either focus mainly on management aspects of SPLs [7], also lack design details, or just provide high level guidelines [16].

In order to model agent features at the Domain Analysis phase, we have adopted some phases of PASSI [5], an agent-oriented methodology. It specifies models with their respective phases for developing MASs, covering all the development process. PASSI integrates concepts from object-oriented software engineering and artificial intelligence, and it follows the guideline of using standards whenever possible. This justifies the use of UML as modeling language. One of the key reasons for choosing PASSI is that the use of a UML-based notation enables the merging of complementary notations proposed in PLUS and PASSI, while establishing a standard for modeling agent and non-agent features.

Instead of using UML for modeling agents in the Domain Design phase, as PASSI proposes, we use an extended version of it, the MAS-ML modeling language [6]. Our focus is to allow the design of agents that follow the belief-desire-intention (BDI) [17] model, whose advantages include: it is relatively mature, and has been successfully used in large scale systems; it is supported by several agent platforms, e.g. Jadex, Jason, JACK and 3APL; and it is based on solid philosophical foundations. As discussed in [6], some important agent-oriented concepts, such as environment, cannot be modeled with UML and the use of stereotypes is not enough because objects and agent elements have different properties and different relationships. As a consequence, other MAS modeling languages do not allow the modeling of some agent concepts. MAS-ML extends the UML meta-model in order to express specific agent properties and relationships. Using its meta-model and diagrams, it is possible to represent the elements associated with a MAS and to describe the static relationships and interactions between these elements.

In summary, we have (i) adopted PLUS notations along all the process; (ii) incorporated three PASSI phases as activities in the Domain Analysis phase; and (iii) used MAS-ML to model agent concepts in the Domain Design phase. In addition, we have proposed (iv) some adaptations to PASSI phases and MAS-ML in order to allow agent variability and agent features traceability; and (v) defined new activities and models to address MAS-PL particularities, as well as specified the sequence and relationship among this activities. The advantage of adopting these specific method fragments that we chose is that, besides providing notations and models for MAS-PLs, they can be integrated with each other. All of them are based on UML, therefore PLUS stereotypes can be incorporated into PASSI and MAS-ML models. PASSI activities are only used at the analysis level, but concepts

identified in these activities (agents and roles) are present in MAS-ML metamodel, so they can be represented in MAS-ML models at the design level.

3.3 Domain Analysis

The Domain Analysis phase defines activities for eliciting and documenting the common and variable requirements of a MAS-PL. It is concerned with the definition of the domain and scope of the MAS-PL, and specifies its common and variable features. This phase comprises two sub-phases: Operational Requirements and Autonomous Requirements.

In the Operational Requirements sub-phase, the systems family is analyzed and its common and variable features are identified therefore defining the MAS-PL scope. Next, requirements are described in terms of use case diagrams and descriptions. The first activity is the Requirements Elicitation, which captures requirements in documents based on interactions with domain specialists and stakeholders. The next activity is the Feature Modeling, which was originally proposed by the FODA [16] method and is the activity of modeling the common and variable properties of concepts and their interdependencies in SPLs. It uses as input the requirements identified in the previous activity. Features are organized into a tree representation, called features diagram, with a specific notation for each variability category (mandatory, alternative and optional). EC features diagram is depicted in Figure 2(a), showing the optional automatic conference suggestion feature. A feature model refers to a features diagram accompanied by additional information such as dependencies among features. It represents the variability within a system family in an abstract and explicit way.

After the Feature Modeling activity, MAS-PL functional features are described in terms of use cases, in the Use Case Modeling activity. First, use cases are identified and described, resulting in both a use case diagram (Figure 2(b)) and use case descriptions. Later, the use case diagram should be refined by: (i) refactoring use cases to provide feature modularization (each use case should correspond to only one feature). It is accomplished by the use of the generalization and the extend relationships; and (ii) adding stereotypes to give variability information (≪*kernel*≫,≪*optional*≫ and ≪*alternative*≫). For instance, the *Suggest Conferences* is an optional use case and extends the *Register Paper* use case, which is part of the MAS-PL kernel. To complete the Operational Requirements, another use case view is modeled to map use cases to features, resulting in the Feature/Use Case Dependency model (Figure 2(c)). Use cases are grouped into features with the UML package notation. These packages are stereotyped with: (i) ≪*common feature*≫ – represents all mandatory features and groups all kernel use cases; (ii) ≪*optional feature*≫ – represents optional features and groups use cases related to a specific optional feature; (iii) ≪*alternative feature*≫ – aggregates alternative features and groups use cases related to a specific alternative feature. Figure 2(c) shows the EC Feature/Use Case Dependency model with two optional features: *Conference Suggestion* and *Deadline Messages*.

The purpose of the Autonomous Requirements sub-phase is to understand better the domain, by modeling autonomy and pro-active concerns with respect

to the current problem domain. This kind of concerns is distinguished because they do not need a user that supervises their execution. Furthermore, they are not well described in use cases, and consequently they need a more precise specification. Agents are an abstraction of the problem space that are a natural metaphor to model pro-active or autonomous behavior. Therefore, it is identified and specified in models in terms of agents and roles.

The domain concepts are captured through the Domain Ontology Modeling activity, in which the MAS-PL domain is modeled through an ontology. The concepts should be modeled taking into account features, by using techniques such as generalization to modularize features. The ontology is represented by UML class diagrams, in which classes and their attributes represent concepts and slots, respectively. Classes of the Ontology Diagram have stereotypes indicating if they are mandatory, optional or alternative, in the same way of use cases. Finally, a model represented by a table that maps concepts to features is created in order to provide feature traceability. This model enables the selection of the appropriate concepts during the application engineering, i.e. if a feature is selected for a product to be derived, the concepts related with this feature according to this map must be present in the product.

In parallel to this activity, features that present pro-active or autonomous behavior (agent features) are identified and specified in models in terms of agents and roles. The Agent Features Identification activity is responsible for such identification. So, a new stereotype (≪*agent feature*≫) is added to the packages of the Feature/Use Case Dependency model to indicate which features are agent features. In Figure 2(c), both optional features are classified as agent features. Our process defines activities for specifying these features, however we do not provide a formal criteria for identifying agent features. Research work in this direction can be seen in [18]. In order to specify the identified agent features, our process incorporates some activities that correspond to some phases of the System Requirements model of PASSI methodology [19]. Next, we briefly describe PASSI phases used in our process and our proposed extensions to them.

- Agent Identification: responsibilities (use cases) are attributed to agents, which are represented as stereotyped UML packages. The use case stereotypes representing the variability, used in the Use Case diagram, are still present. *Our extensions:* only pro-active or autonomous use cases are distributed among agents, while PASSI proposes the realization of all use cases performed by agents; adoption of stereotypes (kernel, alternative or optional); and use of colors to trace features. Figure 2(d) illustrates a partial view of the Agent Identification diagram, in which there are three agents (`DeadlineAgent`, `UserAgent` and `NotifierAgent`). These agents are related with use cases, and relationships between use cases represent the communication between agents.

- Role Identification: agent interactions are explored and expressed through sequence diagrams to identify agent roles. *Our extensions:* diagrams split according to features; use of UML 2.0 frames for representing crosscutting features; use of colors to trace features; and Feature/Agent Dependency model. Figure 2(e) shows a partial view of the *Deadline Messages* Role Identification diagram, in

which class instances represent the role that an agent is playing. In this diagram the `UserAgent` is playing the role `CommitteeMember`. It can be seen a UML 2.0 frame indicating an optional part related with the *Reviewer* feature.

- Task Specification: activity diagrams are used to specify the capabilities of each agent. *Our extensions:* one diagram per agent and feature, while PASSI proposes one diagram per agent; use of UML 2.0 structured activities for representing crosscutting features and use of colors to trace features. Figure 2(g) presents the tasks that the `UserAgent` must perform related with the *Deadline Messages* feature. There is an optional part related with the *Reviewer* feature, delimitated by a UML 2.0 structured activity.

Most of these adaptations are meant to provide variability information in models. Some proposed notations aims at modularizing crosscutting features, which are characterized by having impact in several other features. So, instead of creating new models for them, notations indicate the behavior introduced by this kind of feature. An additional model (Feature/Agent Dependency model) provides support to trace from features to agents (and vice-versa). This model is organized into a tree, whose root represents the MAS-PL, which has children corresponding to agent features. Each feature has agents as its children indicating that these elements must be present in the product being derived if the feature is selected. Agents have roles as children, meaning that the agent plays that roles for a certain feature. Agents and roles may appear more than once in the model, meaning that they will be present in a product if at least one agent feature that depends on them is selected. Figure 2(f) shows a partial view of the Feature/Agents Dependency model illustrating that the optional feature *Deadline Messages* depends on the `UserAgent` agent and the `Chair` role.

3.4 Domain Design

The main purpose of the Domain Design phase is to define an architecture that addresses both common and variable features of a MAS-PL. Based on analysis models, designers must model the MAS-PL architecture, determining how these models, including the variability, are implemented in this architecture. Features modularization must be taken into account during the design of core assets to allow the (un)plugging of optional and alternative features. In addition, there must be a model to map features to design elements providing traceability information.

The Domain Design phase has mainly two parts. First, the MAS-PL architecture is defined and technologies (e.g. frameworks, libraries and agent platforms) that will be used are selected. The MAS-PL architecture is specified in an UML package diagram and defined by its decomposition into subsystems and their components. This helps to reduce the complexity and to allow several design teams to work independently. Choosing appropriate technologies for designing and implementing a MAS-PL is a very important step of its development, because they have an impact on how features are modularized.

On the second part, each feature is statically and dynamically designed in three different activities: Components Modeling, Agents Modeling and Agent Society Modeling. The first activity concerns the design of non-agent features.

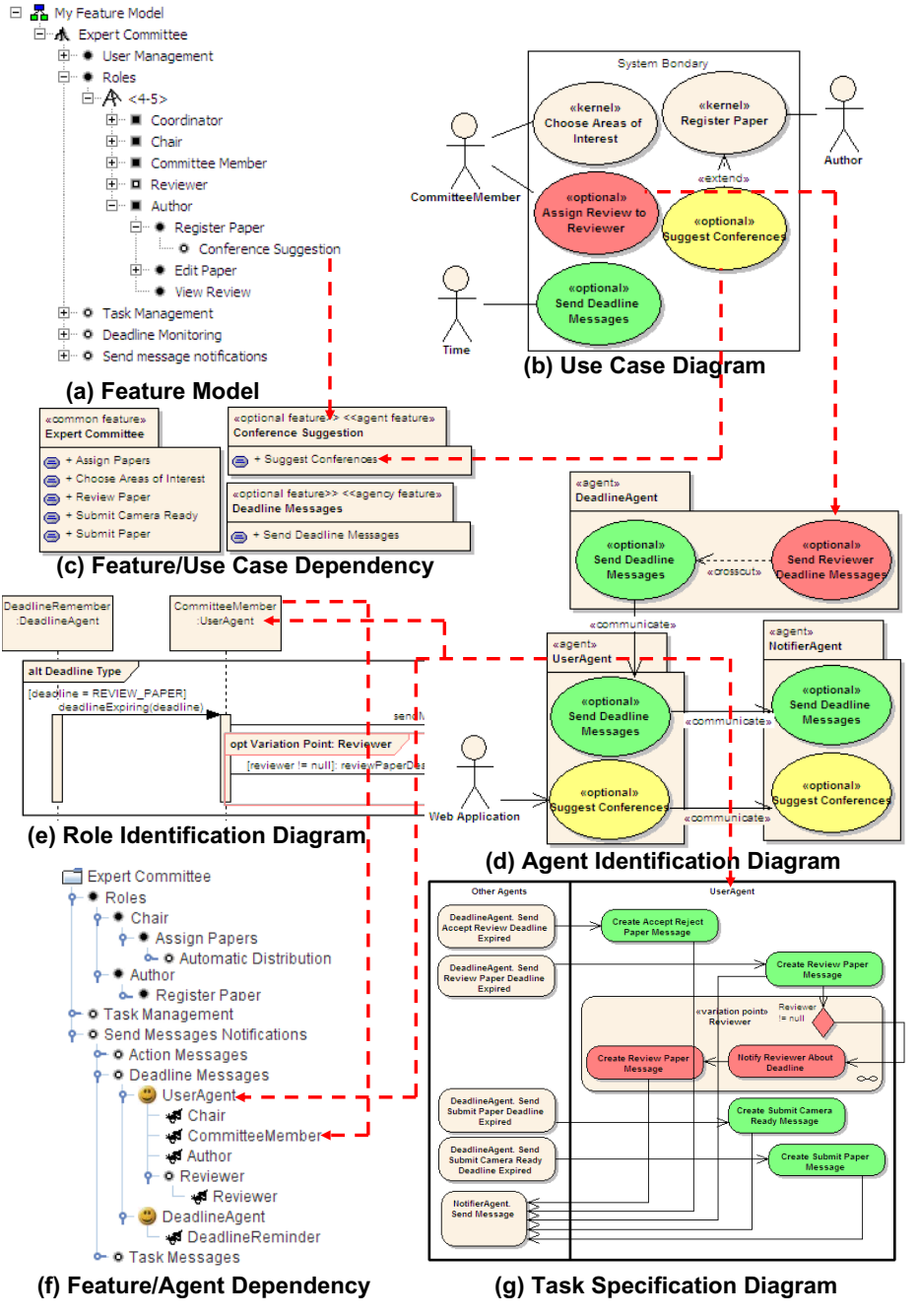


Fig. 2. EC Analysis Work Products (Partial)

This design step is basically provided by the PLUS approach, with the modeling of traditional UML diagrams extended with PLUS stereotypes.

Agent features are modeled in two activities of our process. They are performed in parallel and may contribute with each other. In the Agents Modeling activity, agents with their beliefs, goals and plans are modeled; and in the Agent Society Modeling activity, roles and organizations are modeled. As mentioned in Section 3.2, we use MAS-ML to model agents. Its structural diagrams are the extended UML class diagram and two new diagrams: organization and role. MAS-ML extends the UML class diagram to represent the structural relationships among agent concepts. The organization diagram models system organizations and relationships between them and other system elements. Finally, the role diagram is responsible for modeling relationships between roles defined in organizations.L

To address variability in MAS-ML diagrams, we have adopted four different adaptations: (i) use of $\ll kernel \gg$, $\ll optional \gg$ and $\ll alternative \gg$ stereotypes to indicate variability; (ii) model elements are colored according to the feature they are related to, based on the color assigned for each feature; (iii) model each feature in a different diagram. However, crosscutting features impact in several features, so their specification is spread in other features' diagrams. Although crosscutting features are not modeled in specific diagrams, the use of colors helps to distinguish different model elements related to them; and (iv) the introduction of the capability [20] concept to modularize variable parts in agents and roles. A capability is essentially a set of plans, a fragment of the knowledge base and a specification of the interface to the capability. Capabilities have been introduced into some MASs as a software engineering mechanism to support modularity and reusability. We represent a capability in MAS-ML by the agent or agent role notation with the $\ll capability \gg$ stereotype. An aggregation relationship is used between capabilities and agents, and capabilities and roles.

Figures 3(a), 3(b) and 3(c) show the *Assign Papers* class, role and organization diagrams, respectively. In these diagrams, there are colored elements, which are related with the *Assign Papers* feature. It means that if this feature is selected for a product, the colored elements will be part of the derived product. Most of variable parts in these diagrams are encapsulated into capabilities. For instance, in Figure 3(b) (Role Diagram), most of the colored elements are capabilities that define parts of the `Chair` and `CommitteeMember` roles and a role (`DeadlineMonitor`) that are specific for *Assign Papers* feature. Therefore, the `Chair` role in a product will aggregate a set of capabilities that are related with the selected features of a product.

Agents' dynamic behavior is modeled by means of extended sequence diagrams by MAS-ML. The extended version of this diagram represents the interaction between agents, organizations and environments. The only differences in modeling dynamic behavior for single systems and MAS-PLs are: (i) different features are modeled in different diagrams; and (ii) UML 2.0 frames are used to indicate a behavior related to a crosscutting feature, as it was done in the Role Identification activity (Section 3.3).

Besides modeling agents with appropriate feature modularization and variability notations, the Feature/Agent Dependency model is refined by introducing new agent concepts that were identified in Agents Modeling and Agent Society Modeling activities. This model indicates which design elements should be selected during the product derivation process, i.e. which elements must me present in a product according to a selected set of features. Figure 3(d) shows a partial view of the EC Feature/Agent Dependency model.

3.5 Domain Realization

The purpose of the Domain Realization phase is to implement the reusable software assets, according to the design diagrams. In addition, it incorporates configuration mechanisms that enable the product instantiation process. Two activities compose this phase: Agent Implementation Strategy Description and Assets Implementation.

Implementing software agents is usually accomplished by the use of agent platforms, such as JADE, whose the two main provided concepts are agent and behaviors, and Jadex, which implements the BDI architecture, providing goal, belief and plan concepts. Consequently, there are different ways of implementing agents. In addition, there may be a gap between the design and the implementation, i.e. the concepts adopted to model the MAS-PL may be different of the ones provided by the target implementation platform. So, the goal of the Agent Implementation Strategy Description activity is to define a strategy for implementing agents. For instance, mapping agent concepts used at the design phase (agents, beliefs and plans) to object-oriented concepts (classes, attributes and methods).

In the Assets Implementation activity, designed elements are codified in some programming language. So, the first task of this activity is to implement MAS-PL assets. Different implementation techniques can be used to modularize features in the code, e.g. polymorphism, design patterns, frameworks, conditional compilation and aspect-oriented programming. These techniques are the typical ones used in SPLs. In [21], we presented a quantitative study of development and evolution of the EC MAS-PL, consisting of a systematic comparison between two different versions: (i) one version implemented with object-oriented techniques and conditional compilation; and (ii) the other one using aspect-oriented techniques. Additionally, in [22], we have proposed an architectural pattern to integrate software agents and web applications in a loosely coupled way.

As stated previously, the target implementation platform may force transforming some design elements into platform specific ones. Hence, it is necessary to specify how the design elements were implemented in the selected agent platform for traceability purposes. The Design/Implementation Elements Map is responsible for providing this information. It defines which elements implement which design elements. Figure 3(e) presents a partial view of this model for the EC, showing which classes implemented the `UserAgent`, for instance. With the information provided by the traceability models, it is possible to know how a feature is implemented along the MAS-PL. The Feature/Agent dependency model maps features to design elements, and the Design/Implementation

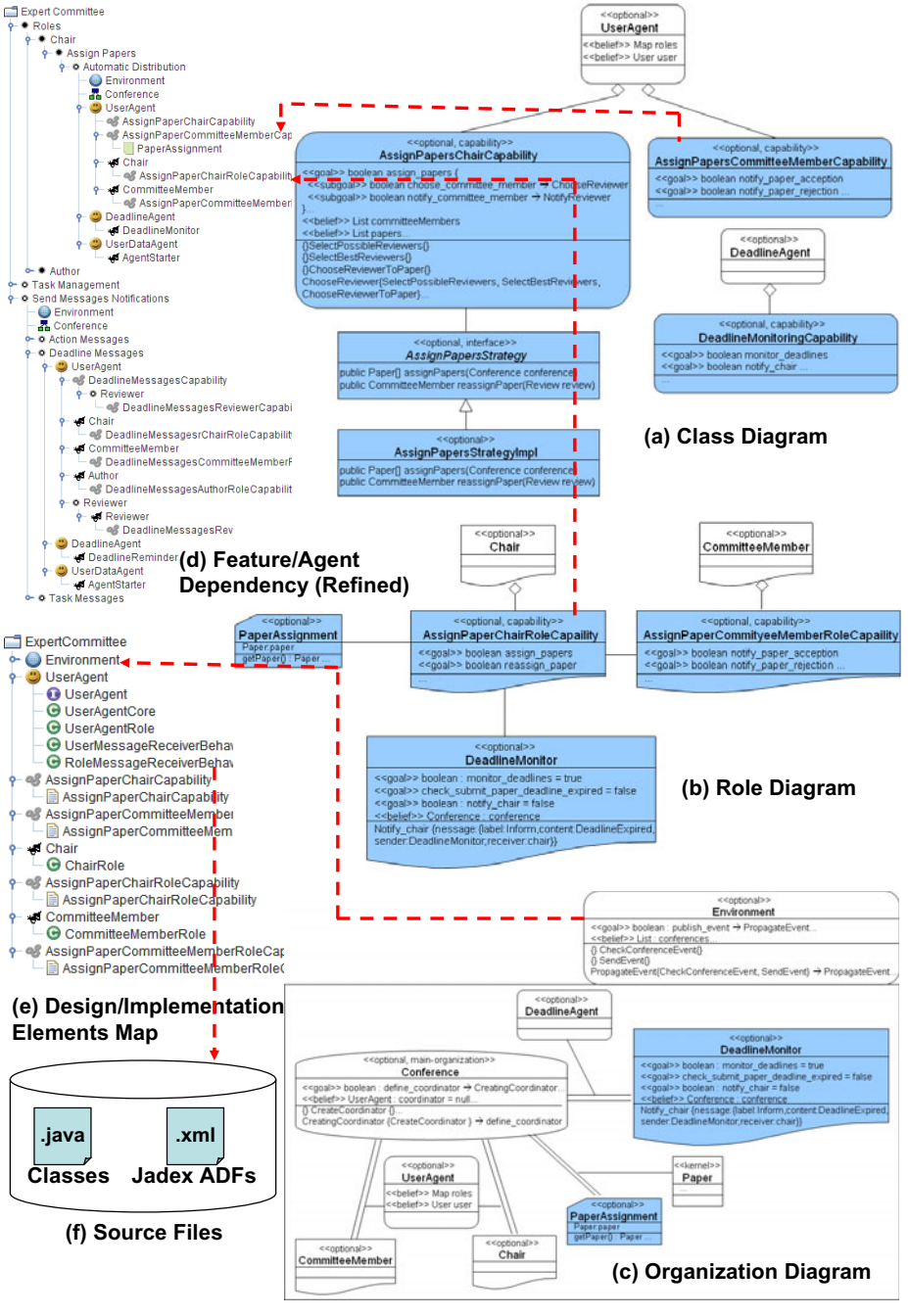


Fig. 3. EC Design Work Products (Partial)

Elements Map provides the information of which design elements are related with implementation elements. Therefore, traceability models provide the configuration knowledge necessary to derive a product given an instance of the feature model. This traceability is illustrated by the red arrows in Figure 3.

4 Conclusions and Future Work

In this paper, we have proposed a domain engineering process for developing MAS-PLs. Using a SPL approach for building MASs allows meeting the need of producing software with mass customization while taking advantage of agent abstraction to model modern software systems that tend to be situated, open, autonomous and highly interactive. Our process was modeled according to SPEM and includes specific activities and work products to address agent features and their traceability, and also provide notations for documenting agent variability. It was also defined based on existing good practices of existing SPL and MAS approaches: PLUS provides notations for documenting variability; PASSI methodology diagrams are used to specify agent features in the Domain Analysis phase; and MAS-ML is the modeling language used in the Domain Design phase. A new contribution of our approach resides in the fact that we completely separate the modeling of agent features. Therefore, it makes it possible to evolve existing systems developed with different technologies to incorporate new features that take advantage of agent abstractions.

Our process has emerged based on the experience of development of two web-based MAS-PLs: the ExpertCommittee (presented in this paper) and the OLIS [22] case study, which is a SPL of web applications that provide personal services to users. In addition, the process was used in a graduate Agent-oriented Software Engineering course at PUC-Rio in order to provide further evaluation of the approach. This experience provided us feedback to improve our process, mainly with notations for crosscutting features. However, we have observed that it would be interesting to investigate how to integrate our approach with application engineering tools in order to allow an automatic product derivation. In addition, the variability information is present in several models, and this may lead to inconsistencies among models during the evolution of the MAS-PL. Therefore, we aim at providing means to manage the consistence among these models.

We are currently developing other case studies to evaluate our process. In addition, we are investigating how model-driven and aspect-oriented approaches can help to model and implement crosscutting features to provide for a better modularization. Finally, we aim at extending our process to address other agent characteristics, such as self-* properties. Future work also includes experimental studies in order to better evaluate our process.

References

1. Zambonelli, F., Omicini, A.: Challenges and research directions in agent-oriented software engineering. JAAMAS 9(3) (2004)
2. Muthig, D., Atkinson, C.: Model-driven product line architectures. In: Chastek, G.J. (ed.) SPLC 2002. LNCS, vol. 2379, pp. 110–129. Springer, Heidelberg (2002)

3. Gomaa, H.: Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures. Addison-Wesley, Reading (2004)
4. Atkinson, C., Bayer, J., Bunse, C., Kamsties, E., Laitenberger, O., Laqua, R., Muthig, D., Paech, B., Wust, J., Zettel, J.: Component-based product line engineering with UML. Addison-Wesley, Reading (2002)
5. Cossentino, M.: From Requirements to Code with the PASSI Methodology, ch. IV. Idea Group Inc., USA (2005)
6. da Silva, V.T., de Lucena, C.J.P.: From a conceptual framework for agents and objects to a multi-agent system modeling language. JAAMAS 9(1-2) (2004)
7. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. Addison-Wesley, Reading (2002)
8. Girardi, R.: Reuse in agent-based application development. In: SELMAS 2002 (2002)
9. Nunes, I., Nunes, C., Kulesza, U., Lucena, C.: Documenting and modeling multi-agent systems product lines. In: SEKE 2008, pp. 745–751 (2008)
10. Peña, J., Hinchey, M.G., Ruiz-Cortés, A., Trinidad, P.: Building the core architecture of a NASA multiagent system product line. In: Padgham, L., Zambonelli, F. (eds.) AOSE VII / AOSE 2006. LNCS, vol. 4405, pp. 208–224. Springer, Heidelberg (2007)
11. Dehlinger, J., Lutz, R.R.: A Product-Line Requirements Approach to Safe Reuse in Multi-Agent Systems. In: SELMAS 2005. ACM Press, New York (2005)
12. Nunes, I., Nunes, C., Kulesza, U., Lucena, C.: Developing and evolving a multi-agent system product line: An exploratory study. In: Luck, M., Gomez-Sanz, J.J. (eds.) AOSE 2008. LNCS, vol. 5386, pp. 228–242. Springer, Heidelberg (2009)
13. Ciancarini, P., Nierstrasz, O., Tolksdorf, R.: A case study in coordination: Conference management on the internet (1998), http://www.cs.unibo.it/cianca/wwwpages/case.ps.gz
14. Nunes, I.: A domain engineering process for mas-pls (2008), http://www.inf.puc-rio.br/~ionunes/maspl/
15. Object Management Group (OMG): Software & Systems Process Engineering Metamodel specification (SPEM) Version 2.0 (2008)
16. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-oriented domain analysis (foda) feasibility study. Technical Report CMU/SEI-90-TR-021, SEI, Carnegie-Mellon University (1990)
17. Rao, A., Georgeff, M.: BDI-agents: from theory to practice. In: ICMAS 1995 (1995)
18. O'Malley, S.A., DeLoach, S.: Determining when to use an agent-oriented software engineering paradigm. In: Wooldridge, M.J., Weiß, G., Ciancarini, P. (eds.) AOSE 2001. LNCS, vol. 2222, pp. 188–205. Springer, Heidelberg (2002)
19. Nunes, I., Kulesza, U., Nunes, C., de Lucena, C.J., Cirilo, E.: A domain analysis approach for multi-agent systems product lines. In: Enterprise Information Systems IV (ICEIS 2009). LNBIP, vol. 24, pp. 716–727 (2009)
20. Busetta, P., Howden, N., Rönnquist, R., Hodgson, A.: Structuring bdi agents in functional clusters. In: Jennings, N.R. (ed.) ATAL 1999. LNCS, vol. 1757, pp. 277–289. Springer, Heidelberg (2000)
21. Nunes, C., Kulesza, U., Sant'Anna, C., Nunes, I., Lucena, C.: On the modularity assessment of aspect-oriented multi-agent systems product lines: a quantitative study. In: SBCARS 2008, pp. 122–135 (2008)
22. Nunes, I., Kulesza, U., Nunes, C., Cirilo, E., Lucena, C.: Extending web-based applications to incorporate autonomous behavior. In: WebMedia 2008 (2008)

Developing Virtual Heritage Applications as Normative Multiagent Systems

Anton Bogdanovych[1], Juan Antonio Rodríguez[2], Simeon Simoff[1], A. Cohen[3], and Carles Sierra[2]

[1] School of Computing and Mathematics, University of Western Sydney, Australia
{anton,simeon}@it.uts.edu.au
[2] IIIA, Artificial Intelligence Research Institute, CSIC, Spanish National Research Council*
{jar,sierra}@iiia.csic.es
[3] Federation of American Scientists, 1725 DeSales Street NW, Washington, DC, USA
acohen@fas.org

Abstract. The majority of existing virtual heritage applications are focused on detailed 3D reconstruction of historically significant sites and ancient artifacts. Recreating the way of life of ancient people is only considered by some researchers, who employ crowd simulation for this task. Existing crowd simulation algorithms are not suitable for modeling complex individual behaviors and role dependent agent interactions with other participants in the Virtual World. To address this problem we suggest treating 3D Virtual Worlds as Normative Multiagent Systems and propose the Virtual Institutions Methodology to be used for design and deployment of Virtual Worlds that require complex interactions involving both humans and autonomous agents. To highlight the usefulness of this approach we illustrate how Virtual Institutions are employed in the development of the Uruk prototype, which integrates 3D Virtual Worlds and Artificial Intelligence in the domain of cultural heritage.

1 Introduction

Non-gaming Virtual Worlds such as Second Life [1] have become an important area of research during the last few years. Many researchers stress the significance of this technology, considering it the next stage of the World Wide Webs evolution (Web 3.0). Gartner has predicted that the majority of the Internet users will be participating in non-gaming Virtual Worlds in the near future [2]. Currently, millions of people spend an average of around twenty hours a week in various Virtual Worlds [3]. Furthermore, studies in South Korea have indicated that the majority of Koreans prefer 3D Virtual Worlds to television [4].

Two promising application domains for non-gaming Virtual Worlds are cultural heritage and education. In heritage domain 3D graphics is used to reconstruct lost sites of high historical significance. In education the interest in Virtual

* Thanks to projects JC2008-00337, TIN2006-15662-C02-01, CONSOLIDER CSD2007-0022.

M.-P. Gleizes and J.J. Gomez-Sanz (Eds.): AOSE 2009, LNCS 6038, pp. 140–154, 2011.

Worlds is particularly strong in relation to history. In both of these domains researchers are normally focused on reconstructing destroyed or damaged buildings (e.g. the Roman Coliseum). While such an approach allows for the examining of architectural details of the heritage site in three dimensions, it still does not help a general observer to understand how this site has been used in the past.

Populating history or heritage oriented Virtual Worlds with avatars that behave similar to the ancient citizens of the reconstructed places has a potential to provide visitors with a more engaging experience. Using human experts to control the avatars to simulate the behavior of ancient citizens is very costly, while employing Artificial Intelligence for this task (which is a much more affordable option) hasn't received appropriate attention from research. The majority of researchers that are working on populating virtual heritage sites with avatars employ so-called virtual crowds [5] for this task. Such crowds normally consist of a large number of autonomous agents (represented as avatars) dressed appropriately for the selected period of time and appearing as local citizens of the reconstructed area. The state of the art in combining crowd simulation and 3D heritage reconstruction can be observed via the example outlined in [5]. Here, a 3D reconstruction of the ancient City of Pompeii is "made alive" using a large number of avatars that walk around the city avoiding collisions. While providing a visitor with some understanding about the appearance of the ancient people, this approach poorly elaborates on specific behavioral characteristics of those people. The agents employed in [5] do not use the objects in the environment and are not engaged into historically authentic interactions.

Through the use of Virtual Institutions technology [6] we intend to bring virtual heritage to a new level by making it more dynamic and interactive. Instead of just having virtual crowds walking around the city we suggest populating virtual heritage sites with autonomous agents that reenact the most typical daily activities of the reconstructed society. Creating such agents is quite a challenging task as the degree of interaction is pretty high, the agents have to depend on other agents, play different roles, synchronize their activities with other agents and even solve some tasks in a teamwork manner while actively using the objects in the virtual environment. One of our research hypotheses is that for an autonomous agent to be able to demonstrate similar complexity of actions as ancient humans, the complexity of agent's environment must be reduced. This assumption is based on the suggestion made by Russell & Norvig that the agents ability to successfully participate in some environment and extend its intelligence there is highly dependent on the complexity of this environment [7]. It is suggested that situating the agent in a fully observable, deterministic and discrete environment helps the agent to tackle the famous frame problem of AI [8].

As an example of a fully observable, deterministic and discrete environment, we consider Virtual Institutions [6] (previously known as 3D Electronic Institutions), which are 3D Virtual Worlds with normative regulation of participants' interactions. In Virtual Institutions the environment is formalized in terms of norms of acceptable behavior of participants, interaction protocols and role flow of participants. Every agent has access to this formalization, which helps it to

reason about its own actions and the actions of other participants (either humans or agents) as well as to understand the consequences of these actions. The Virtual Institutions technique that we employ for such environment formalization is based on Electronic Institutions [9] widely used in Multiagent Systems for structuring the interactions of the agents participating in open systems.

Overall, the Virtual Institutions approach to the development of applications for virtual heritage is to treat 3D Virtual Worlds as Normative Multiagent Systems. Our work builds on top of the research published in [10]. The original *methodology* presented in [10] has been applied to a real world problem and, as a result of this, *was revised*. The updated methodology includes *new steps* and also features a *detailed explanation of the first two steps*. The most significant *contribution* of this paper is providing the *evaluation* of the Virtual Institutions methodology by developing a prototype in the domain of cultural heritage.

The remainder of the paper elaborates on the details of the contributions presented above. In Section 2 we describe the concept of Virtual Institutions, the corresponding methodology and technology. Section 3 provides the motivation for using Virtual Institutions in the domain of cultural heritage, illustrates the application of the Virtual Institutions methodology to this domain and outlines all the development steps. Finally, Section 4 presents some concluding remarks.

2 Virtual Institutions

We consider Virtual Institutions [6] being a new class of Normative Virtual Worlds, that combine the strengths of 3D Virtual Worlds and Normative Multiagent Systems, in particular, Electronic Institutions [9]. In this "symbiosis" the 3D Virtual World component spans the space for visual and audio presence, and the normative component takes care of enabling the formal rules of interactions among participants. Through the normative component a Virtual World is separated into a number of logical spaces (scenes). Only participants playing particular roles are admitted to a scene and can change the state of this scene. Once admitted the participants should follow the interaction protocol specified for each scene and are unable to violate this protocol. The institution doesn't take away an agent's autonomy by forcing it to act in a specific manner, but restricts it by prohibiting to violate the institutional rules and enforcing a particular interaction protocol in every scene.

As a benefit of this approach the agents' possible actions are clearly defined and comprise a finite set. Moreover, applying such restrictions helps to formalize the environment so that every action of any participant (human or agent) and its consequences can be sensed and formalized for every agent.

2.1 Virtual Institutions Methodology

The Virtual Institutions methodology [10], outlined in Fig. 1 a), covers the entire development process. Its application requires 7 steps to be accomplished:

1. Eliciting Specification Requirements.
2. Specification of an Electronic Institution.
3. Verification of the specification.
4. Automatic Generation of the corresponding 3D environment (if needed).
5. Annotation of the Electronic Institution specification with components of the 3D Virtual World.
6. Integrating the 3D Virtual World into the institutional infrastructure.
7. Enabling Implicit Training

One of the key contributions of this paper is the description of the methods that should be utilized on steps 1 and 2 of the methodology – as outlined in Fig. 1 b). Therefore, here we elaborate on these steps, while referring to the initial work published in [10] for the detailed description of other steps.

Step 1. Eliciting Specification Requirements. This step of the methodology aims at producing the Software Requirements Document where the key activities, roles of the participants, and basic scenarios are outlined. Upper part of the Fig. 1 b) presents the details of this step. The key task here is enforcing institution designers to provide an answer to the following core issues regarding a virtual institution: where is it situated (environment), what is it designed for (social goals), how it is expected to succeed (abstract norms), which social roles are required to enact it and how can participants interact (ontology).

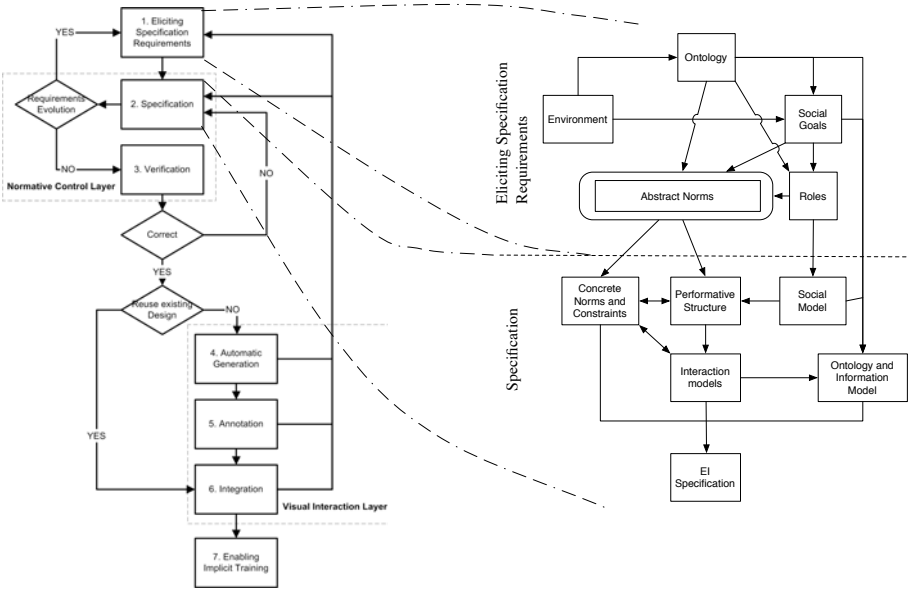


Fig. 1. Virtual Institutions Methodology

1.1 Environment. Firstly, a system architect must consider *where* the resulting system is to be deployed, in which *physical* and *social environment* it has to perform its activities, what are the factors that will affect its behaviour that it can or cannot control? We take the stance that all objects in the environment are beyond the *complete* control of the institution to be defined. Agents will interact with that environment by *observing* and by *affecting* those objects in some way. In order to cope with the mentioned lack of control we will assume that all of them are agentified and that the perception and action upon them is always dealt with in the context of a dialogue with the agent that wraps them.

1.2 Ontology. To define institutional norms we must express normative behaviour in some language, which in turn requires the definition of some *ontology*, namely the definition of what agents are going to talk about. The ontology can be represented as a set of concepts and relations among them. The ontology has also to accommodate the objects already found when specifying the environment.

1.3 Social goals. Typically institutions are regulated environments where humans interact to attain some global objectives, which we call *social goals*. An institution aims at continuously satisfying its social goals and will not allow any agent to behave in a way that prevents from attaining them. The specification of social goals (e.g. in first-order logic or natural language) must employ the ontology defined in the previous step. The output of this step is a list of social goals.

1.4 Roles. Next, we identify the *roles* (patterns of agent behaviour) in the institution, which are determined to a large extent by the social goals. What the institution aims at achieving indicates which capabilities and behaviours are needed. Roles, at this stage, are a set of identifiers with some associated capabilities expressed as the message types (i.e. ontology terms) the role is capable of dealing with. The output of this step is a list of roles and their capabilities.

1.5 Abstract norms. The idea of an abstract norm is that of expressing a *generic restriction* on how the agents incarnating the identified roles should behave. The roles and the ontology previously analysed provide the language needed to write the expressions of the abstract norms. To avoid any confusion, the designer should clearly distinguish between abstract norms and social goals because abstract norms are assumed to refer to the *behaviour of the agents*, and not to the ultimate *purpose* of the institution, which is represented by the social goals. Abstract norms cannot prevent agents from satisfaction of any social goal. The output of this step is a list of abstract norms.

Step 2. Specification. This step establishes the regulations that govern the behavior of the participants. This process is supported by ISLANDER [11], which permits to specify most of the components graphically, hiding the details of the formal specification language and making the specification task transparent. The details of this methodology step are outlined as a number of substeps in the lower part of Fig. 1 b) that we detail below.

2.1 Social model. The initial task of the system architect is enriching the role model by adding further information to turn it into a *social model*. A distinction between internal and external roles must be made. Roles whose raison-d'être is

to support the achievement of the social goals and to enforce the norms will be marked as *internal* roles and the rest as *external* roles. The following relationships between the roles relevant for the institution are specified: (i) hierarchy, (ii) ssd, static separation of duties, and (iii) dsd, dynamic separation of duties.

2.2 Performative structure. Next the definition of the set of dialogical activities permitted for different roles entails a sequence of substeps:

1. Starting from the abstract norms, the designer must define a list of scenes (activities in the institution) along with their participating roles.
2. For each scene in the list, its creation conditions (i.e. which role(s) initiates the scene), and whether it can be multiply enacted or not must be specified.
3. Based on the abstract norms, the designer gathers together scenes into a performative structure by specifying: (i) the flow of agent roles, namely which roles from which scenes can get into other scenes; (ii) the role change policy, namely whether agents are allowed to change roles when moving out of a scene into another scene. The result is a graph connecting scenes whose edges are labelled with expressions encoding the role flow and role change policies.

2.3 Interaction model. Once the performative structure is defined the designer must associate some *interaction model* to each scene, namely a specification of the dialogue in the scene (v.g. through a finite state machine). An interaction model must contain the conversation states, the illocutions exchanged between agents that permit transitions between conversation states, and the constraints that restrict these transitions. Moreover, this substep also requires to set the minimum and maximum number of agents playing each role allowed in the scene as well as the conversation states where new agents are permitted to join and participating agents are permitted to leave. Sometimes, when going through this step, we may identify new roles and thus a revised version of the social model is intermingled with this substep. Moreover, we may also identify refinements in the ontology that help us better express the constraints restricting transitions. This substep and the next one are very interrelated.

2.4 Concrete norms and constraints. A further refinement still applies to the performative structure and its scenes' interaction models. Back to the abstract norms, we may find that some of them can be translated into simple *constraints* that will limit agents in two ways: (i) by not permitting them to say certain things (by adding a constraint into some interaction model); and (ii) by not permitting them to move to a certain scene (by adding a constraint into the performative structure). Some abstract norms, however, will require a more sophisticated representation because the effect of an agent action (illocution) implies that certain other action(s) is *actually* done.

2.5 Ontology and information model. Finally, from the initial ontology definition and the concrete messages as expressed in the different interaction models of the institution, the *ontology* can be completed. Further analysis of the social model, and the specification of the performative structure and its scenes' interaction models helps to complete the identification of the *attributes* (along with their types) of: roles, objects in the environment, scenes, and the insitution itself.

Step 3. Verification. One of the advantages of the formal nature of the Virtual Institutions methodology is that the specification produced on the previous step can be automatically verified for correctness by ISLANDER [11]. The tool verifies the scene protocols, the role flow among the different scenes and the correctness of norms (see [10] for details). If any errors are found, developers must return to step 1 to correct those. If the specification contains no errors, there are two options. If the 3D Visualization of the environment is already created (reuse of the existing design) then the developers may skip the next step and continue with Step 4. Otherwise, the generation step, Step 3, should be executed.

Step 4. Automatic Generation. In some cases it is desirable to generate the initial skeleton of the Virtual World from the specification. In such cases the Virtual World can be generated automatically using the method presented in [12]. The resulting Virtual World has to be annotated on Step 5.

Step 5. Annotation. The Specification defines the rules of the interaction and has nothing to say about the appearance of the specified elements. On step 5 the specification is enriched with appearance related graphical elements. These additional elements include textures and 3D Objects like plants, furniture elements etc. This step of the methodology does not usually require the involvement of the system architects and should rather be executed by designers and software developers. After this step the user can return to Steps 1 and 2 to refine the specification requirements or the specification itself or can continue with Step 6.

Step 6. Integration. On the integration step the execution state related components are specified. This includes the creation of the set of scripts that control the modification of the states of the 3D Virtual Worlds and mapping of those scripts to the messages, which change the state of the Electronic Institution. After this step the user can return to Steps 1 and 2 to refine the specification requirements or the specification itself or can continue with Step 7.

Step 7. Enabling Implicit Training. Having the complete formalization of the agent environment makes it possible to use imitation learning as a key technique for achieving human-like behavior of the agents. The institutional specification forms the basis for the decision graph of the agent, where possible illocutions become the nodes of this tree (see [13]). On Step 7 for each of those nodes we specify whether implicit training is conducted or not.

2.2 Deployment

A virtual institution is enabled by a three-layered architecture presented by conceptually (and technologically) independent layers [6]. The Normative Control Layer employs AMELI [14] to regulate the interactions between participants by enforcing the institutional rules. The Communication Layer causally connects the institution dimensions with the virtual world [6]. The Visual Interaction Layer (currently supported by Second Life [1]) visualizes the Virtual World.

3 City of Uruk: Virtual Institutions in Cultural Heritage

Uruk [15] is a joint research project between the University of Western Sydney and the Federation of American Scientists. The aim of the project is to recreate the ancient city of Uruk from the period around 3000 B.C. in the Virtual World of Second Life letting the history students experience how it looked like and how its citizens behaved in the past. The Virtual World of Second Life provides a unique collaborative environment for history experts, archaeologists, anthropologists, designers and programmers to meet, share their knowledge and work together on making the city and the behavior of its virtual population historically authentic.

3.1 Significance of Uruk

Uruk was an ancient city located in present day Iraq (circa 250 km south of Baghdad). Many historians consider Uruk being one of the first human built cities on Earth. By 2900 B.C. Uruk is believed to be one of the largest settlements in the world and one of the key centers of influence of the Sumerian culture. Uruk played a major role in the invention of writing, emergence of urban life and development of many scientific disciplines including mathematics and astronomy.

3.2 The Prototype

The prototype aims at showing how to enhance the educational process of history students by providing them with a possibility to immerse into an accurate replica of the daily life of the ancient citizens of Uruk and gain quick understanding of the advance of technological and cultural development of ancient Sumerians. Ultimately, the students may become part of the virtual society and will have to interact with agents and other humans to solve the assigned tasks.

The 3D reconstruction of the city was produced within the Virtual World of Second Life based on the results of archeological excavations and available written sources. Both modeling of the city and programming of the virtual humans populating it were conducted under the supervision of subject matter experts.

We have selected fishermen daily life of ancient Uruk to illustrate how Virtual Institutions can enable immersive experience in the life of ancient societies. We created four agents that represent members of two fishermen families (see Fig. 2 and Fig. 3). Each family consists of a husband and wife. Every agent has a unique historically authentic appearance and is dressed appropriately for 3000 B.C.

Each of the agents enacts one of the four social roles. Agent Fisherman1 plays the “SpearOwner” role. He is the young male fisherman. He possesses the fishing spear and is capable of catching the fish with it. He and his brother also jointly own a fishing boat. His daily routine consists of waking up on the roof, having a morning chat with fisherman 2, fishing, bringing the fishing gear back home and climbing back on the roof to sleep.

Wife1 Andel enacts the “WaterSupplier” role. She is the young wife of Fisherman1, who is responsible for collecting water from the well. Her daily routine consists of waking up on the roof, collecting the water from the well, doing house

Fig. 2. Fisherman Family 1: Fisherman1 Andel and Wife1 Andel

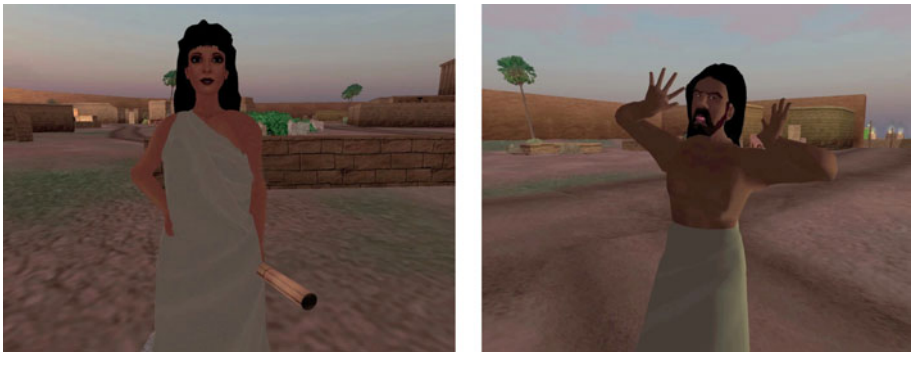


Fig. 3. Fisherman Family 2: Wife2 Jigsaw and Fisherman2 Jigsaw

work and climbing back on the roof to sleep there. As any other typical fisherman wife in Uruk she does not have any recreation time and is constantly working.

Agent Fisherman2 is the older brother of Fisherman1 playing the social role "BoatOwner". He lives with his wife in the separate house next to his brother. Both families are very close and spend most of their day together. Fisherman2 possesses a fishing basket and paddles for rowing the fishing boat. His daily routine consists of waking up on the roof, having a morning chat with Fisherman1, fishing, bringing the fishing gear back home and climbing on the roof to sleep.

Wife2 Jigsaw, the wife of Fisherman2, plays the "FireKeeper" role. She is older than Wife1 and, therefore, is the key decision maker for controlling the integrity of both households. She makes fishing baskets and trades them for other household items. Starting the fire and preparing food are her direct responsibilities. Her daily routine consists of waking up on the roof, starting a fire for cooking, routine house work and climbing back on the roof to sleep there.

The agents literally "live" in the virtual world of Second Life. Their day is approximately 15 minutes long and starts with waking up on the roof of the building – Fig. 4 a). Although, most of the buildings in Uruk had ventilation holes the temperatures inside (especially during summer) could become quite

Fig. 4. The City of Uruk Prototype

unpleasant and most of the citizens would prefer sleeping on the roof top in the evening, where it would have been much cooler. The wives wake up first to collect some water from the well and prepare breakfast for their husbands. The husbands start their day with a morning chat while waiting for the breakfast to be prepared.

After breakfast the fishermen would collect their fishing gear and walk towards the city gates – Fig. 4 b). Outside the gates on the river bank they find their boat which they will both board and start fishing. One of the agents would be standing in the boat with a spear trying to catch the fish and the other agent would be rowing. Fig. 4 c) illustrates the fishing process.

After fishing, the men exit the boat, collect the fishing basket and spear and bring them back to their homes. This daily cycle is then continuously repeated with slight variations in agent behavior.

3.3 Development of the Prototype

The Virtual Institutions Methodology was employed for the development of the prototype. Here we provide a step-by-step description of how it was followed.

Step 1: Eliciting Specification Requirements. In order to come up with realistic specification requirements the system developers have conducted joint meetings with 2 subject matter experts. The decision on the roles of participants, scenes they can participate in and interaction protocols have been identified through conversations with subject matter experts. It was agreed that having four agents in the system is sufficient for the first prototype. The roles of these agents are: two fishermen and two fishermen wives. The identified scenes that must be present in the environment are: Home1, Home2, FirePlace, Well, Chat and Fishing. Below we provide a fragment of the resulting Software Requirements Document produced as the result of applying this step of the methodology.

Environment contains: boat, house, well, fire, spear, fish, river, fishing basket.

Ontology Relationships. River has fishes, baskets contain fishes, men own boats.

Social goals. Daily provision of food to cater for the city's needs.

Abstract norms. Fishermen must go daily fishing for a limited number of hours; The number of fishing baskets per household is limited to avoid overexploitation of natural resources; Women are in charge of housework; Men alone cannot go fishing; Fishing is not allowed at night; Hoarding is prohibited and punished; All men under some age are obliged to fish;

Social model. Fishermen are in static separation of duties with wives.

Performative structure. Fishermen wake up, chat, fish, eat, sleep, and back again. Wives wake up, set up a fire, collect water, cook, sleep, and back again.

Interaction model. Description of fishing. At least 2 men on a boat. When both men on a boat, its state changes to "sailing". While sailing the fishermen holding a spear can throw it to catch fishes. If a spear gets a fish, the fisherman can remove the fish from the spear and put it in the fishing basket. The action of throwing the spear is an illocution, catching a fish is treated as an event generated by the spear. The rest treated as properties of roles, spear, and fishing baskets.

Concrete norms and constraints. Men are obliged to fish under the age of 35; one fishing basket per household max; fishing permitted between dawn and dusk; at least 2 men in a boat; if hoarding (more than one fishing basket), fishermen can be prohibited to go sailing for a week.

Step 2: Specification. Based on the information obtained at Step 1 the formal specification of the underlying Electronic Institution was produced with ISLANDER [11]. The first step in specifying the system corresponds to identifying the social roles of the actors and the relationships among these roles. Fig. 5 illustrates the role hierarchy for our case.

Fig. 6 shows the Performative Structure of the specification. The nodes of this graph feature the identified scenes and the arcs define the role flow of

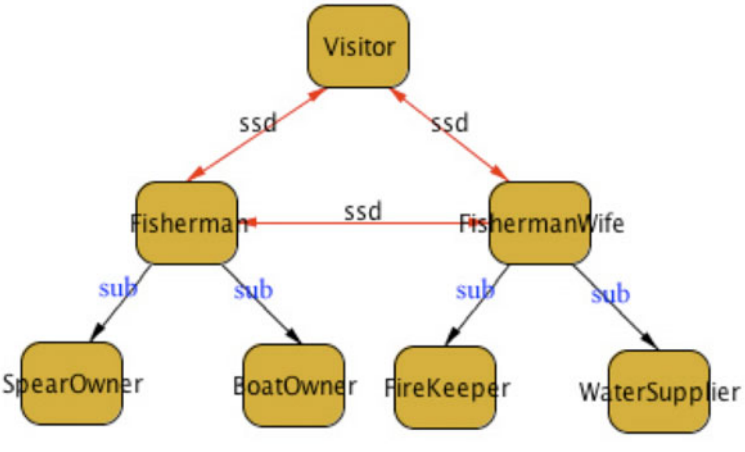


Fig. 5. Role Hierarchy in the Uruk System

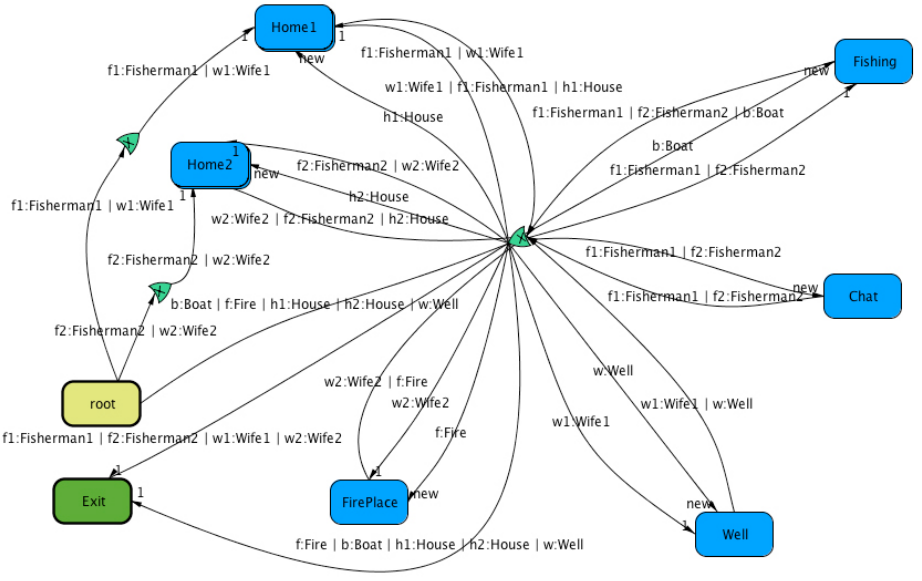


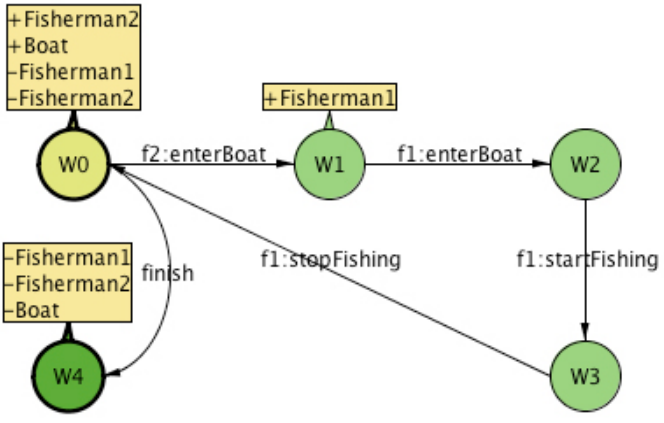
Fig. 6. Uruk Performative Structure

participants amongst these scenes. Arcs labelled with "new" define that the participant with the role name appearing above the arc is initializing the scene and no other participants can enter it before the initialization occurs. The "bumpy" appearance of Home1 and Home2 suggests that these scenes permit multiple execution (which in our case corresponds to having different floors in the building, so that the agents can sleep inside their homes and on its roof).

The "root" and "exit" scenes are not associated with any patterns of behavior and simply define the state of entrance and exit of participants into the institution. Apart from "root" and "exit" each of the scenes present in the Performative Structure is associated with a Finite State Machine defining the interaction protocol for the participants that are accepted into the scene. In order to change the scene state the participant has to perform an illocutionary act (by sending a message to the institutional infrastructure).

To give an example of the scene formalization Fig. 7 outlines the scene protocol for the Fishing scene and outlines the illocutions that change the states of this scene. The scene protocol here defines in which sequence agents must perform the actions, at which point can they join and leave the scene and what should they do to change the scene state. In Virtual Institutions we consider every action that changes the state of the institution being a speech act (text message). Every action (i.e. grabbing an object or clicking on it) a participant performs in a Virtual World is automatically transformed into a speech act similar to those presented in the lower part of Fig. 7.

Once the scene is initialized its initial state becomes "W0". While the scene is in this state Fisherman2 and Boat can join the scene and both Fisherman1 and



f2:enterBoat = (request (?f2 Fisherman2) (all all) enter)
f1:enterBoat = (request (?f1 Fisherman1) (all all) enter)
f1:startFishing = (request (!f1 Fisherman1) (all all) start)
f1:stopFishing = (inform(!f1 Fisherman1) (all all) finish)
finish = (inform(?b Boat) (all all) finish)

Fig. 7. Fishing Scene: Interaction Protocol

Fisherman2 can leave the scene (the "Boat" joins the scene automatically once it is activated). Fisherman1 can only enter the scene when it evolves to state "W1" as the result of Fisherman2 informing all the scene participants that he has successfully entered the boat. This happens once the corresponding avatar boards the boat. After entering the boat Fisherman1 must notify the participants about this fact so that the scene can evolve to "W2". For this to happen the corresponding agent must send the "f1:enterBoat" illocution to Fisherman2 and Boat. Then Fisherman1 may request to start fishing, which would bring the scene into "W3". The result of this is the change of the boat property from "standing" to "afloat", Fisherman2 will start rowing and the boat object will move. While in "W3" the only action that can be done is informing all the participants by Fisherman1 that fishing is finished. When the fishing is finished Fisherman2 must return the boat to the initial poistion, park it there, drop the paddles, take the fishing basket and exit the boat. Fisherman1 will also have to exit the boat. No participants can leave the scene in this state and must wait until the scene evolves to "W0". While the scene is in "W0" again the Boat object may stop the scene by sending the "finish" illocution. Doing so would mean deactivating the scene and making it impossible for the participants to join it and act on it (no participant will be able to sit inside the boat).

Similar to the Fishing scene the interaction protocols have to be specified for other scenes present in the Performative Structure. We would like to point out

that the scene protocol does not define how the actual fishing should take place, but simply provides the key states within the scene so that the agents can have a formal understanding of the performed actions.

Step 3: Verification. This step provides the possibility to ensure that the resulting specification is correct. ISLANDER provides an automatic way of verifying the correctness of the specification as well as the error notification system. Until the specification is error free it must be further revised in ISLANDER.

Step 4. Automatic Generation. In our case one of the system requirements was having a historical accurate reconstruction of the city based on the results of archaeological excavations. For such a case automatic generation is not appropriate and, therefore, Step 4 and 5 have been skipped. The 3D design of the city of Uruk was created manually under supervision of subject matter experts.

Step 5. Annotation. In our case the annotation step of the methodology is not required as the 3D environment was created manually.

Step 6. Integration. At this step the dynamic objects are supplied with the corresponding LSL (Linden Scripting Language) scripts to enable interaction dynamics. To be able to maintain the causal connection between the institutional state and the state of the Virtual World the actions that change the state of the Virtual World are mapped to illocutions that change the institutional state.

Another important part of the annotation process is to associate each scene with the corresponding subspace within the Virtual World in order to be able to control the movement of the agents between scenes. In case of the Virtual World of Second Life the solution for making this correspondence is to use invisible objects tightly enclosing each given scene.

Programming the agents so that they can act in the environment and use the institutional specification for the decision making is also done at this step.

Step 7. Enabling Implicit Training. This functionality is currently missing in the described prototype. In the future this step will be used to let the agents learn state transitions from human experts controlling the avatar.

The result of following the methodology is an immersive environment simulating the culture of the ancient city of Uruk along the following dimensions: environment, agents, artifacts and institutions. As our experiments have confirmed [16], each of these dimensions of the given culture can be successfully learned by the end users who tested our prototype.

4 Conclusion

We have shown that virtual heritage applications that involve autonomous agents reenacting the way of life of ancient people should be treated as Normative Multiagent Systems. For design and deployment of such Virtual Worlds we developed the Virtual Institutions Methodology. This methodology is supplied with a set of tools that facilitate the design, development and execution of such environments. We would like to stress that, to our knowledge, Virtual Institutions is the

first methodology that is specifically concerned with the development of Virtual Worlds with normative regulation of interactions. To evaluate the methodology we have applied it to the implementation of the Uruk prototype visualizing the behavior of citizens of ancient Mesopotamia, 3000 B.C.

References

1. Linden Lab: Second Life, http://secondlife.com
2. Gartner Inc: Gartner Says 80 Percent of Active Internet Users Will Have A "Second Life" in the Virtual World by the End of 2011 (2007), http://www.gartner.com/it/page.jsp?id=503861 (visited 23.07.2007)
3. Hunter, D., Lastowka, F.G.: To kill an avatar. Legal Affairs (July 2003)
4. Weinstein, J., Myers, J.: Same principles apply to virtual world expansion as to China and other new markets. Media Village (November 28, 2006), http://www.mediavillage.com/jmr/2006/11/28/jmr-11-28-06/
5. Maïm, J., Haegler, S., Yersin, B., Mueller, P., Thalmann, D., Van Gool, L.: Populating Ancient Pompeii with Crowds of Virtual Romans. In: 8th International Symposium on Virtual Reality, Archeology and Cultural Heritage - VAST (2007)
6. Bogdanovych, A.: Virtual Institutions. PhD thesis, UTS, Australia (2007)
7. Russel, S., Norvig, P.: Artificial Intelligence a Modern Approach. Prentice Hall/Pearson Education International, Upper Saddle River, New Jersey, USA (2003)
8. Dennett, D.C.: Cognitive wheels: The frame problem of AI. In: Minds, Machines, and Evolution: Philosophical Studies. Cambridge University Press, Cambridge (1984)
9. Esteva, M.: Electronic Institutions: From Specification to Development. PhD thesis, Institut d'Investigació en Intelligència Artificial (IIIA), Spain (2003)
10. Bogdanovych, A., Esteva, M., Simoff, S., Sierra, C., Berger, H.: A methodology for 3D Electronic Institutions. In: Proceedings of AAMAS 2007 Conference, Honolulu, Hawaii, USA, pp. 346–348. ACM, New York (2007)
11. Esteva, M., de la Cruz, D., Sierra, C.: ISLANDER: an Electronic Institutions editor. In: First International Conference on Autonomous Agents and Multiagent systems, Bologna, pp. 1045–1052. ACM Press, New York (2002)
12. Drago, S., Bogdanovych, A., Ancona, M., Simoff, S., Berger, H., Sierra, C.: From Graphs to Euclidean Virtual Worlds: Visualization of 3D Electronic Institutions. In: Dobbie, G. (ed.) Australasian Computer Science Conference (ACSC 2007), Ballarat Australia. CRPIT, vol. 62, pp. 25–33. ACS (2007)
13. Bogdanovych, A., Simoff, S., Esteva, M.: Virtual institutions: Normative environments facilitating imitation learning in virtual agents. In: Prendinger, H., Lester, J.C., Ishizuka, M. (eds.) IVA 2008. LNCS (LNAI), vol. 5208, pp. 456–464. Springer, Heidelberg (2008)
14. Esteva, M., Rosell, B., Rodriguez-Aguilar, J.A., Arcos, J.L.: AMELI: An Agent-Based Middleware for Electronic Institutions. In: Proceedings of AAMAS 2004 Conference, pp. 236–243. IEEE Computer Society, Los Alamitos (2004)
15. Uruk Project, http://www-staff.it.uts.edu.au/~anton/Research/Uruk_Project
16. Bogdanovych, A., Rodriguez, J.A., Simoff, S., Cohen, A.: Virtual Agents and 3D Virtual Worlds for Preserving and Simulating Cultures. In: Ruttkay, Z., Kipp, M., Nijholt, A., Vilhjálmsson, H.H. (eds.) IVA 2009. LNCS, vol. 5773, pp. 257–271. Springer, Heidelberg (2009)

Part IV
State-of-the-Art Survey

Modelling with Agents

Estefanía Argente[1], Ghassan Beydoun[2], Rubén Fuentes-Fernández[3], Brian Henderson-Sellers[4], and Graham Low[5]

[1] Universidad Politécnica de Valencia, Spain
eargente@dsic.upv.es
[2] University of Wollongong, Australia
beydoun@uow.edu.au
[3] Universidad Complutense de Madrid, Spain
ruben@fdi.ucm.es
[4] University of Technology Sydney, Australia
brian@it.uts.edu.au
[5] University of New South Wales, Australia
G.Low@unsw.edu.au

Abstract. Modelling is gaining relevance in Agent-Oriented Software Engineering (AOSE) because of two main reasons. Firstly, the conceptual frameworks have reached a level of maturity that makes it reasonable to devote effort to seek a consensus in modelling languages, including tool support. Secondly, the influence of model-driven engineering emphasizes the potential value of having models at the core of development processes. This survey analyzes these changes in AOSE modelling languages along three dimensions. The semantic dimension refers to the concepts considered in the languages. The syntactic dimension covers the technical means by which languages are defined. The operational dimension regards the use of these languages, considering both their support and acceptance. The overall context for this discussion is the comparison of several modern AOSE approaches.

Keywords: Multi-agent system, Model, Modelling language, Comparison, Semantics, Syntax, Use.

1 Introduction

Agent-Oriented Software Engineering (AOSE) [28] regards modelling as a key task in the development of Multi-Agent Systems (MASs). It provides a variety of supporting elements for it, such as notations, guidelines and tools. However, the use of models in AOSE has been limited by three main factors. Firstly, the paradigm has experienced difficulties to reach agreement on the concepts to be considered [6]. These have been continuously evolving and have frequently been ill-defined in the literature. Secondly, there has been a lack of suitable techniques to handle both modelling languages (MLs) and models [38]. This has resulted in conceptual frameworks frequently evolving more quickly than their formal representations, thus hampering tool support for them. Thirdly, it is hard to

M.-P. Gleizes and J.J. Gomez-Sanz (Eds.): AOSE 2009, LNCS 6038, pp. 157–168, 2011.

capture application domains in models and consistently transform them into running code [19].

Seeking to overcome these limitations, AOSE research is working in two main directions. Firstly, the field is pursuing consensus on what the relevant concepts for modelling a MAS are as well as their definitions. This trend can be observed in the research on unification frameworks [1,6,33][1], mappings between notational sets [36] and toolset comparisons [15]. The result of these efforts is expected to be a widely accepted, stable and precisely defined conceptual framework for MAS modelling. Secondly, AOSE is increasingly adopting Model-Driven Engineering (MDE) [38] approaches, as it is seen in the growing number of authors who consider models to be at the core of an AOSE development process [28]. This has impact on the way we may choose to define a ML: with an increasing incorporation of metamodels and their use to generate different development artefacts; and with a growing use of automated transformations. These kinds of approaches are intended to favour a closer alignment between MLs, their support tools and the code produced from models.

In order to analyze the evolution along these two research lines, this paper considers the different aspects in the definition of a ML [29]: the semantics define the concepts present in the ML and their meaning; the syntax comprehends the abstract syntax, i.e. the representation of these concepts in terms of entities, relationships, attributes and constraints, and the concrete syntax, i.e. the actual representation of the elements of the abstract syntax, for instance with text or graphical notations. A third aspect considered here is the context in which a ML is used, including documentation, processes, support tools and diffusion.

The study of the previous aspects considers several recent and illustrative AOSE MLs. Some of them are stand-alone languages while others are an integrated part of an AOSE methodology. For convenience, we will use the term "ML" irrespective of whether the ML is published independently, e.g. FAML [6], ANote [9] and MAS-ML [39], or as part of a methodology, e.g. Adelfe [4], INGENIAS [37] and Prometheus [34,35]. The evolution and inter-relationships of all these "MLs" are depicted in Fig. 1. It can be seen that significant influences other than within the research team are few and far between. Older approaches such as Adelfe and PASSI have had some influence but the two most obvious beneficiaries of earlier work are the two projects aiming to create a convergence, FAML and the "Unifying MMM" [5]. Many methodologically-based MLs also use elements of UML [32], AUML [3,31] or AML [8] — three languages developed in the standards arena rather than directly in the context of a specific methodology. UML Profiles are also used, for instance in AOR's (*Agent-Object-Relationship*) ML [41]. Agent modelling is also supported in i* [43], especially for the requirements engineering stage.

The MLs chosen for analysis in this survey are, in alphabetical order, Adelfe [4], FAML [6], INGENIAS [37], O-MaSE [14], PASSI [10], Prometheus [34], SODA [30] and Tropos [23] – those which, in the subjective judgement of the

[1] Although the simple juxtaposition of several metamodels [5] without due regard to ontology unification is unlikely to be successful in the long term.

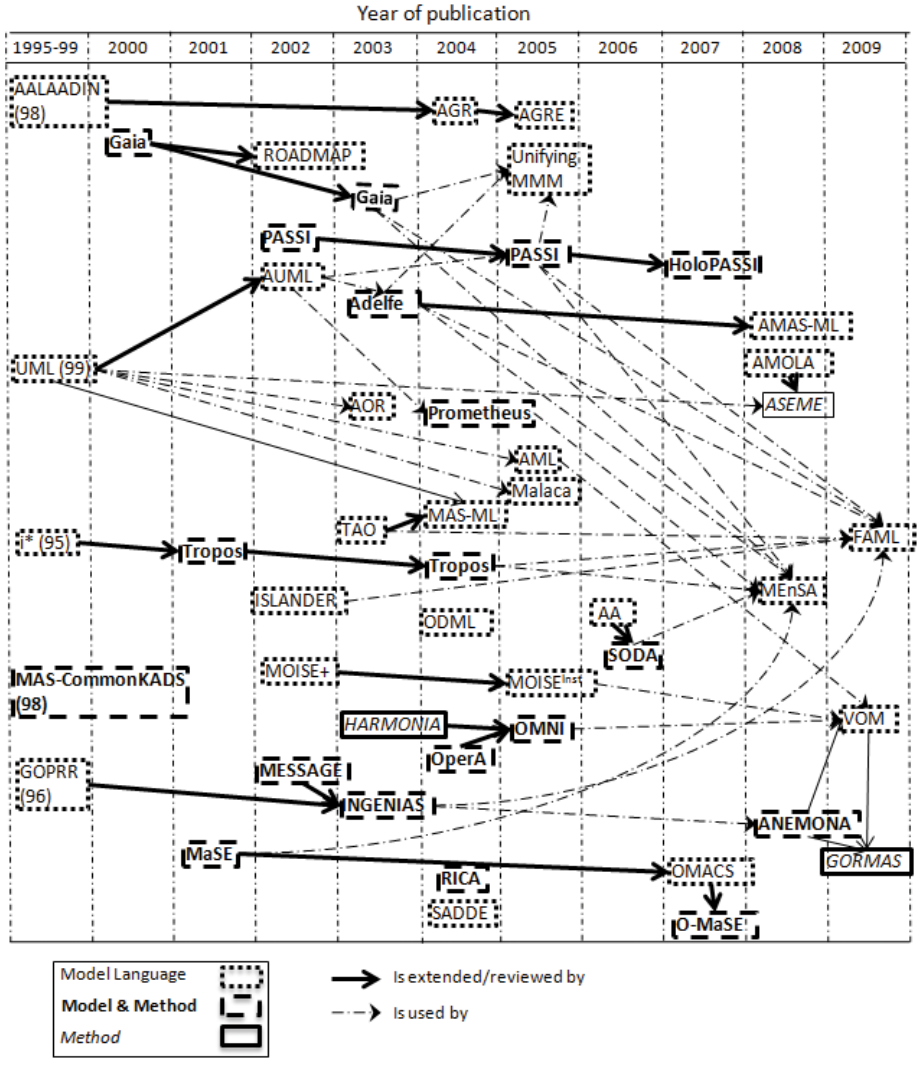


Fig. 1. Temporal evolution of AOSE MLs. [Note that, for space limitation reasons, not all elements in this diagram are cited or discussed in the text.]

authors of this paper, have not only a history but a potential evolution. We specifically exclude MLs for mobile agents – see for example the UML/AUML extensions discussed in [25].

The rest of the paper is organized as follows. The next three sections analyze AOSE MLs along the three previously described dimensions: semantic (see Sect. 2), syntactic (see Sect. 3) and use (see Sect. 4). Finally, Sect. 5 discusses some trends in agent-oriented modelling based on the analysis in the previous sections.

2 Semantic Perspective

The semantics of a ML [29] describes the intended understanding that its designers try to communicate to their users. This review adopts a translational semantics, so it considers the semantics of the different languages in terms of the translation of their concepts to a common conceptual framework.

The chosen conceptual framework consists of three levels [6]: the *system* level considers the overall organization of the MAS; the *agent* level the perspective of individual agents; and the *environment* level refers to the context of the MAS beyond its boundaries. The analysis of these levels can be organized by applying the dimensions identified for organizations in [11]: *structural*, which describes the entities in a given level; *functional*, which details the capabilities of the elements; *dynamical*, which considers the interactions between the elements and their effects; and *normative*, which defines mechanisms used by the society to influence its members' behaviour and applies only to the system and agent levels.

At the system level, structural concepts include, for instance, groups, roles, their dependencies and compatibility, and the organization topology. Functional concepts deal with service descriptions, tasks, system requirements (both functional and non-functional) and system goals. Dynamic aspects describe agent interactions (using protocols and message schemata) and system evolution (mainly dealing with descriptions of role enactment and specifications of mental state). Finally, normative features are usually described using concepts of deontic logics, such as obligations and prohibitions.

At the agent level, the structural dimension applies to agent descriptions and specifications of their mental states. Its functional dimension deals with the agents' autonomy by means of plan and action specifications. The dynamic dimension considers both agent communication (i.e. features for agent interaction) and situatedness (i.e. relation to its environment). Normative issues cover norm acceptance together with processes to reason about norms.

The environment level includes artefacts in the environment and their relationships. It usually describes the functional perspective with methods of these artefacts. The dynamic features indicate how to represent the changes in the state of the artefacts and their interactions.

Table 1 summarizes the semantic comparison. It considers the suitability of the different approaches to describe the levels and dimensions previously discussed more than specific concepts.

The analysis shows a bias to the system level with all the approaches providing primitives for its different aspects. Only the normative issues are commonly disregarded despite their relevance in the literature [42], although there are exceptions. OMNI [17] explicitly defines norms and contracts, and O-MaSE [14] defines policies for role enactment, organization behaviour and reorganization. Other recent MLs, such as MOISE+ [26] or MEnSA [1], cover the normative dimension by means of rules describing permissions and obligations.

The description of the internals of agents has complete coverage for the structural and functional levels in these approaches. However, only five of them consider the description of how these elements provide the dynamic behaviour,

Table 1. Semantic features of AOSE MLs

Semantic features	Modelling languages								
	Adelfe	FAML	INGE-NIAS	O-MaSE	OMNI	PASSI	Prome-theus	SODA	Tropos
- System level									
Structural	√	√	√	√	√	√	√	√	√
Functional	√	√	√	√	√	√	√	√	√
Dynamic	√	√	√	√	√	√	√	√	√
Normative					√				
- Agent level									
Structural	√	√	√	√	√	√	√	√	√
Functional	√	√	√	√	√	√	√	√	√
Dynamic	√	√	√	√			√		
Normative									
- Environment level									
Structural	√	√	√	√		√	√	√	√
Functional	√	√	√	√		√	√	√	
Dynamic								√	

including specific support to represent elements such as triggering conditions, exchanged information or post-conditions of the execution of tasks. The issue of how agents internalize the norms of the society is disregarded by all the approaches. Though there is relevant research in this issue [7,12], it seems not to have been integrated in the MLs in this survey.

The environment of a MAS is another key aspect of the paradigm [42], but these MLs barely consider it. Most of the approaches just include a set of artefacts, so agents act on the environment through their methods and perceive it through the events they produce. Only SODA [30] goes further and adds rules for the internal behaviour, use, inspectability and malleability (i.e. dynamic adaptation) of these artefacts.

3 Syntactic Perspective

The syntactic dimension of the study of MLs [29] considers the ways of defining them, with aspects such as formalisms, symbols or complexity of definitions. AOSE uses a variety of approaches for this purpose, summarized in Table 2.

Designers can describe the syntax of a ML in three ways. They can use formal approaches with different kinds of grammars (e.g. textual, iconic or graphs) and/or metamodels. The increasing relevance of MDE tends to emphasize metamodels as the main mechanism for this purpose [24]. The second approach is to use documents to explain the ML, with text and examples. Thirdly, tools provide information about MLs through their use. The only reviewed languages that use grammars are OMNI [17] and Tropos (but only for Formal Tropos [20]), which are related to logics. All the other approaches use metamodels and/or tools, and have MLs that are graph-based with a graphical notation. In the case of Tropos,

Table 2. Syntactic features of AOSE MLs

Syntactic features	Modelling languages								
	Adelfe	FAML	INGENIAS	O-MaSE	OMNI	PASSI	Prometheus	SODA	Tropos
- Definition means									
Document									√
Tool	√		√	√	√	√	√		√
Grammar					√				√
Metamodel	√	√	√	√		√			√
- Aspects									
Abstract syntax	√	√	√	√	√	√	√	√	√
Concrete syntax	√		√	√	√		√		√
Tool information	√		√	√		√	√		√
- Basic primitives									
N-ary relations	√	√	√	√	√	√			√
Arbitrary attributes									
Entity hierarchy	√	√	√	√		√			√
Relation hierarchy									
Modularization	√		√				√	√	
Diagrams	√	√	√	√	√	√	√	√	√
- Extensibility									
Language	√		√			√			
Process adaptation			√						
Tool adaptation			√						
- Processing									
Usability	√		√	√		√	√		√
Scalability	√		√		√			√	

it has both metamodel and grammar, but they define partly disjoint MLs. Most approaches only have descriptions of their ML in research papers (i.e. journals and proceedings). These are usually very condensed for space reasons, which can make it difficult to understand them. Tropos [23] is the only approach that offers a complete explanatory documentation of its ML, although others like Adelfe [4], INGENIAS [37] and Prometheus [34] have partial descriptions.

All the reviewed approaches regard the abstract syntax of their ML, but only some of them define a specific concrete syntax. For instance, PASSI [10] depicts its concepts with an UML-like notation similar to class diagrams; FAML [6] uses detailed English text organised in a dictionary defining and tabulating the metamodel concepts — current work is aimed at developing a specific concrete syntax with a graphical notation; SODA [30] defines a tabular notation. Among the languages with their own iconic concrete syntax are Adelfe [4] and Prometheus [35]. All the approaches with support tools also store tool-related information, such as users' preferences or workspaces. Nevertheless, this information does not appear in the formal definition of the languages except in INGENIAS [22].

There are relevant differences in the basic primitives present in these MLs, with some features commonly accepted while others only appear in specific

languages. For instance, few languages allow the use of inheritance to extend their concepts and, when they allow it, it is only in a limited way. This is the case of INGENIAS [37], where users can only extend agents, or PASSI [10], which uses inheritance for ontologies. Another aspect is the availability of mechanisms to structure information as a means to deal with the complexity of real specifications. These elements do not represent concepts of the paradigm (e.g. an agent grouping goals, capabilities and knowledge), but structures with a common objective or describing an aspect of the system (e.g. the subsystem that manages load balancing or the static view of a MAS). Diagrams are widely accepted for this purpose, and in particular certain types of them [27]. Withal, only four MLs out of the nine studied here offer some modularization mechanism for artefacts in diagrams. SODA [30] only allows zooming in and out diagrams with different levels of detail, while the remaining three approaches provide general packages.

Extensibility considers how approaches deal with the needs of evolving their ML. This is an aspect that mature software engineering (SE) approaches include to allow domain customization [2]. Few AOSE MLs include suitable mechanisms for this purpose. Only Adelfe [4], INGENIAS [37] and PASSI [10] have some kind of limited inheritance between entities. Other extensions require guidelines on how to modify the definition of the ML and its related processes and tools. Only INGENIAS [22] has this information available.

The definition of a ML facilitates processing its specifications when it is regular and of limited size. This is to some extent the case of Adelfe [4], O-MaSE [14], PASSI [10] and Prometheus [35]. The availability of a rich documentation also facilitates processing, as happens with Tropos [23], which has a document describing the complete language, and INGENIAS [37], which has specific documents about processing. Scalability refers to the organization of complex specifications to facilitate understanding and processing. It is strongly related to the availability of elements such as diagrams, packages or imports of partial specifications. Only OMNI [16] offers a truly scalable approach due to its neat separation of concerns and the explicit traceability between elements in its specifications.

4 Operational Perspective

The utility of a ML is not only a matter of providing suitable concepts and representations for them. In the context of modern SE, engineers demand complementary resources that help them in learning and applying a ML. Table 3 summarizes some of these aspects.

In the context of MLs, development processes describe modelling tasks, guidelines to use their models and participant roles. Most of the reviewed MLs have such related processes. The only exception is FAML [6], which currently only defines the metamodel of its language. The processes related with the other MLs focus on the analysis and design stages. Only OMNI [16] and Tropos [23] include suitable support for requirements elicitation (e.g. specific modelling primitives and verification techniques), though others, such as Adelfe [4] and INGENIAS [37], include at least use cases for this task. Despite the claim of supporting

Table 3. Operational features of AOSE MLs. [Note that FAML is the only pure ML, unless the other approaches that are complete AOSE methodologies].

Operational features	Modelling languages								
	Adelfe	FAML	INGE-NIAS	O-MaSE	OMNI	PASSI	Prome-theus	SODA	Tropos
- Process									
Elicitation					√				√
Analysis	√	√	√	√	√	√	√	√	√
Design	√		√	√		√	√		√
Implementation			√	√					
Testing	√		√	√					√
- Tools									
Modelling	√		√	√	√	√	√		√
Model manipulation	√		√						
Coding			√						
Verification	√		√						√
Documentation			√						
- Documentation									
Manuals	√		√	√		√	√		√
Case studies	√		√	√		√	√		√
Projects									
- Acceptance									
External research	√		√				√		√
Industrial use									

implementation (i.e. coding) of many approaches, their support is usually poor. Only INGENIAS includes a detailed description on how to perform the MAS implementation together with abstractions and tools needed to facilitate it; nevertheless, others such as Adelfe, O-MaSE [14,21] (as it extends MaSE [13]) and Prometheus [34,35] provide some less elaborated support. Finally, Adelfe, INGENIAS, O-MaSE and Tropos include some guidelines to test artefacts of the development, such as documentation, requirements or code.

The complexity of current developments also requires tools to accomplish their tasks. Most of the MLs considered here have related tools [15], as all but FAML are linked to development methodologies. The functionality and extensibility of these tools vary widely: all of them support modelling, but few allow using the models as the basis for other tasks such as verification or coding. Some of them can be extended to cover these needs, though it is not a trivial task. For instance, this is the case of the tools of Adelfe [4], INGENIAS [37], O-MaSE [21], OMNI [18] and Tropos [23]. The extension commonly consists in programming new modules for the tool, a task that only INGENIAS documents properly.

Documentation is another aspect to consider about MLs. Newcomers find it difficult beginning with the agent paradigm, as it is not mainstream SE and its approaches are frequently on the edge of research, using tools and techniques that are not common. Thus, having a proper documentation is vital to get adoption of these approaches. This is an aspect usually disregarded by AOSE

works beyond research papers. However, the research papers are necessary condensed so they do not constitute a suitable documentation for someone without an extensive background in AOSE. These readers need documents such as users' manuals, extensive case studies, step-by-step process descriptions or getting started problems, either as printed or web documents or embedded in the tools help. Those methodologies that have been more used outside their original groups are those with better documentation. Two approaches deserve here special mention: Prometheus is the only one with a published book exclusively devoted to development using it [34]; INGENIAS with extensive documentation and training material at its website `http://grasia.fdi.ucm.es/`. Withal, this documentation cannot be considered complete when comparing it with more widespread approaches of SE such as the UML [32].

A final consideration is whether the efforts of the different approaches have been accepted outside their originating groups. Here, the publicly available results raise criticism about their dissemination. Only three approaches consistently report published research with them outside their original groups, and none an industrial adoption beyond research projects, to the best of our knowledge.

5 Discussion and Conclusions

The previous sections have offered an overview of the state and trends in MLs for AOSE. Three levels have been analyzed: semantic, syntactic and operational.

The semantic analysis draws a landscape of convergence between approaches, with a growing agreement about the key concepts of the paradigm. Differences between approaches mainly come from their focus on specific types of MASs or particular aspects of them. Efforts to define a unified ML for AOSE are able to cover increasingly wide portions of different MAS models, as happens with FAML and MEnSA. Only the normative and environment aspects of MASs still lack suitable support in most general modelling approaches.

The syntactic analysis shows a growing acceptance of graphical MLs based on graphs. Even approaches with a text-bias, such as those based on logics, provide (or will provide) some graphical representation of their models. Related to this trend, metamodels become the predominant means to define the abstract syntax of a ML [24]. Nevertheless, there are relevant differences between approaches about the basic modelling primitives to include in the abstract syntax, and the use of concrete syntaxes and tool-specific information. The analysis also points out important limitations in the mechanisms to customize MLs, which make them rigid for non-standard uses, that is, those not similar to their case studies.

The operational analysis covers the use of these MLs. It shows an increasing effort to provide MLs and development processes able to support industrial projects. However, the publicly available results still show that AOSE approaches need to make stronger efforts to gain adoption outside their originating groups, in both other research groups and industry.

Aggregating the results of the three dimensions, three maturity levels can be distinguished among approaches. A first group comprehends works that are at

the conceptual level, providing useful insights in theoretical aspects. This group includes FAML, OMNI and SODA. A second group includes approaches that constitute work in progress, as they are exploring the application of research lines in development. Here appear approaches such as O-MaSE and PASSI. The final group comprehends several methodologies that are mature and complete enough to be applicable in industrial developments of limited size, though they need to overcome shortcomings in their processes, documentation and tools. This group includes methodologies as Adelfe, INGENIAS, Prometheus and Tropos.

Acknowledgments. The authors acknowledge support from the Spanish Council for Science and Innovation under grants CONSOLIDER-INGENIO 2010 CSD2007-00022 and TIN2008-06464-C03-01 (Agent-based Modelling and Simulation of Complex Social Systems, SiCoSSys), and from the Australian Research Council under grant DP0878172 (Ontology-based agent-oriented development methodologies).

References

1. Ali, R., Bryl, V., Cabri, G., Cossentino, M., Dalpiaz, F., Giorgini, P., Molesini, A., Omicini, A., Puviani, M., Seidita, V.: MEnSA Project - Methodologies for the Engineering of complex Software systems: Agent-based approach. Technical Report 1.2, UniTn (2008)
2. Arlow, J., Neustadt, I.: UML 2 and the Unified Process. Addison-Wesley, Reading (2005)
3. Bauer, B., Müller, J.P., Odell, J.: Agent UML:A Formalism for Specifying Multiagent Software Systems. International Journal of Software Engineering and Knowledge Engineering 11(3), 207–230 (2001)
4. Bernon, C., Camps, V., Gleizes, M.P., Picard, G.: Engineering Adaptive Multi-Agent Systems: The ADELFE Methodology. In: Henderson-Sellers, B., Giorgini, P. (eds.) Agent-Oriented Methodologies, pp. 172–202. Idea Group Publishing, USA (2005)
5. Bernon, C., Cossentino, M., Gleizes, M.P., Turci, P., Zambonelli, F.: A Study of Some Multi-agent Meta-models. In: Odell, J.J., Giorgini, P., Müller, J.P. (eds.) AOSE 2004. LNCS, vol. 3382, pp. 62–77. Springer, Heidelberg (2005)
6. Beydoun, G., Low, G., Henderson-Sellers, B., Mouratidis, H., Gómez-Sanz, J.J., Pavón, J., Gonzalez Perez, C.: FAML: a Generic Metamodel for MAS development. IEEE Transactions on Software Engineering 35(6), 841–863 (2009)
7. Boella, G., van der Torre, L.: Regulative and Constitutive Norms in Normative Multiagent Systems. In: 9th International Conference on Principles of Knowledge Representation and Reasoning (KR 2004), pp. 255–265. AAAI Press, Menlo Park (2004)
8. Cervenka, R., Trencansky, I.: AML. The Agent Modeling Language. Birkhäuser Verlag AG, Basel (2007)
9. Choren, R., Lucena, C.: The ANote Modeling Language for Agent Oriented Specification. In: Choren, R., Garcia, A., Lucena, C., Romanovsky, A. (eds.) SELMAS 2004. LNCS, vol. 3390, pp. 198–212. Springer, Heidelberg (2005)
10. Cossentino, M., Gaglio, S., Sabatucci, L., Seidita, V.: The PASSI and Agile PASSI MAS Meta-Models Compared with a Unifying Proposal. In: Pěchouček, M., Petta, P., Varga, L.Z. (eds.) CEEMAS 2005. LNCS (LNAI), vol. 3690, pp. 183–192. Springer, Heidelberg (2005)

11. Criado, N., Argente, E., Julián, V., Botti, V.: Designing Virtual Organizations. In: Demazeau, Y., Pavón, J., Corchado, J.M., Bajo, J. (eds.) 7th International Conference on Practical Applications of Agents and Multi-Agent Systems (PAAMS 2009). Advances in Intelligent and Soft Computing, vol. 55, pp. 440–449. Springer, Heidelberg (2009)
12. Dastani, M., Dignum, V., Dignum, F.: Organizations and normative agents. In: Shafazand, H., Tjoa, A.M. (eds.) EurAsia-ICT 2002. LNCS, vol. 2510, pp. 982–989. Springer, Heidelberg (2002)
13. DeLoach, S.A.: The MaSE Methodology. In: Bergenti, F., Gleizes, M.P., Zambonelli, F. (eds.) Methodologies and Software Engineering for Agent Systems. Multiagent Systems, Artificial Societies, and Simulated Organizations., vol. 11, pp. 107–125. Springer, USA (2004)
14. DeLoach, S.A.: Engineering Organization-Based Multiagent Systems. In: Garcia, A., Choren, R., Lucena, C., Giorgini, P., Holvoet, T., Romanovsky, A. (eds.) SELMAS 2005. LNCS, vol. 3914, pp. 109–125. Springer, Heidelberg (2006)
15. DeLoach, S.A., Padgham, L., Perini, A., Susi, A., Thangarajah, J.: Using three AOSE toolkits to develop a sample design. International Journal of Agent-Oriented Software Engineering 3(4), 416–476 (2009)
16. Dignum, V., Dignum, F., Meyer, J.J.: An Agent-Mediated Approach to the Support of Knowledge Sharing in Organizations. The Knowledge Engineering Review 19(2), 147–174 (2005)
17. Dignum, V., Vázquez-Salceda, J., Dignum, F.: OMNI: Introducing Social Structure, Norms and Ontologies into Agent Organizations. In: Bordini, R.H., Dastani, M.M., Dix, J., El Fallah Seghrouchni, A. (eds.) PROMAS 2004. LNCS (LNAI), vol. 3346, pp. 181–198. Springer, Heidelberg (2005)
18. Dignum, V., Okouya, D.: Operetta: A prototype tool for the design, analysis and development of multi-agent organizations. In: 7th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2008), Demo track, pp. 1677–1678. IFAAMAS (2008)
19. France, R., Rumpe, B.: Model-driven development of complex software: A research roadmap. In: Future of Software Engineering 2007, in 31st International Conference on Software Engineering (ICSE 2007), pp. 37–54. IEEE Computer Society, Los Alamitos (2007)
20. Fuxman, A., Kazhamiakin, R., Pistore, M., Roveri, M.: Formal Tropos: language and semantics, version 1.0. Technical report, University of Trento and IRST (2003)
21. Garcia-Ojeda, J.C., DeLoach, S.A.: Robby: agentTool III: From Process Definition to Code Generation. In: 8th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2009), vol. 2, pp. 1393–1394. IFAAMAS (2009)
22. García-Magariño, I., Fuentes-Fernández, R., Gómez-Sanz, J.J.: A Framework for the Definition of Metamodels for Computer-Aided Software Engineering Tools. Information and Software Technology 52(4), 422–435 (2010)
23. Giorgini, P., Kolp, M., Mylopoulos, J., Castro, J.: Tropos: A Requirements-Driven Methodology for Agent-Oriented Software. In: Henderson-Sellers, B., Giorgini, P. (eds.) Agent-Oriented Methodologies, pp. 20–45. Idea Group Publishing, USA (2005)
24. Gonzalez-Perez, C., Henderson-Sellers, B.: Metamodelling for Software Engineering. J. Wiley and Sons, Chichester (2008)
25. Hachicha, H., Loukil, A., Ghedira, K.: MA-UML: a conceptual approach for mobile agents modelling. International Journal of Agent-Oriented Software Engineering 3(2-3), 277–305 (2009)

26. Hannoun, M., Boissier, O., Sichman, J.S., Sayettat, C.: MOISE: An Organizational Model for Multi-Agent Systems. In: Monard, M.C., Sichman, J.S. (eds.) SBIA 2000 and IBERAMIA 2000. LNCS (LNAI), vol. 1952, pp. 156–165. Springer, Heidelberg (2000)
27. Henderson-Sellers, B.: Consolidating diagram types from several agent-oriented methodologies. Submitted to International Journal of Agent-Oriented Software Engineering (2010)
28. Henderson-Sellers, B., Giorgini, P. (eds.): Agent-oriented methodologies. Idea Group Publishing, USA (2005)
29. Kleppe, A.: A Language Description is More than a Metamodel. In: 4th International Workshop on Software Language Engineering (ATEM 2007), pp. 1–9 (2007)
30. Molesini, A., Omicini, A., Denti, E., Ricci, A.: SODA: A roadmap to artefacts. In: Dikenelli, O., Gleizes, M.-P., Ricci, A. (eds.) ESAW 2005. LNCS (LNAI), vol. 3963, pp. 49–62. Springer, Heidelberg (2006)
31. Odell, J., Parunak, H.V.D., Bauer, B.: Extending UML for agents. In: Wagner, G., Lespérance, Y., Yu, E. (eds.) Agent-Oriented Information Systems Workshop, in the 17th National Conference on Artificial Intelligence, pp. 3–17 (2000)
32. Object Management Group (OMG): OMG Unified Modeling Language (OMG UML), Superstructure. Version 2.2. OMG (2009)
33. Object Management Group (OMG): Agent Metamodel and Profile (AMP). OMG Initial submission, OMG document ad/2009—08–04 (2009)
34. Padgham, L., Winikoff, M.: Developing Intelligent Agent Systems: A Practical Guide. J. Wiley and Sons, Chichester (2004)
35. Padgham, L., Winikoff, M.: Prometheus: A Practical Agent-Oriented Methodology. In: Henderson-Sellers, B., Giorgini, P. (eds.) Agent-Oriented Methodologies, pp. 107–135. Idea Group Publishing, USA (2005)
36. Padgham, L., Winikoff, M., DeLoach, S., Cossentino, M.: A Unified Graphical Notation for AOSE. In: Luck, M., Gomez-Sanz, J.J. (eds.) AOSE 2008. LNCS, vol. 5386, pp. 116–130. Springer, Heidelberg (2009)
37. Pavón, J., Gómez-Sanz, J.J., Fuentes-Fernández, R.: The INGENIAS Methodology and Tools. In: Henderson-Sellers, B., Giorgini, P. (eds.) Agent-Oriented Methodologies, pp. 236–276. Idea Group Publishing, USA (2005)
38. Selic, B.: The pragmatics of model-driven development. IEEE Software 20(5), 19–25 (2003)
39. da Silva, V.T., Choren, R., de Lucena, C.J.P.: MAS-ML: a multiagent system modelling language. International Journal of Agent-Oriented Software Engineering 2(4), 382–421 (2008)
40. Silva, V., Garcia, A., Brandão, A., Chavez, C., Lucena, C., Alencar, P.: Taming Agents and Objects in Software Engineering. In: Garcia, A.F., de Lucena, C.J.P., Zambonelli, F., Zhang, S.-W., Castro, J. (eds.) Software Engineering for Large-Scale Multi-Agent Systems. LNCS, vol. 2603, pp. 103–136. Springer, Heidelberg (2003)
41. Taveter, K., Wagner, G.: Towards Radical Agent-Oriented Software Engineering Processes Based on AOR Modelling. In: Henderson-Sellers, B., Giorgini, P. (eds.) Agent-Oriented Methodologies, pp. 277–316. Idea Group Publishing, USA (2005)
42. Weiss, G.: Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence. The MIT Press, Cambridge (2000)
43. Yu, E.S.K.: Modelling strategic relationships for process reengineering. Ph.D Thesis. Department of Computer Science, University of Toronto, Toronto (1995)

A Survey on the Implementation of Agent Oriented Specifications

Ingrid Nunes[1], Elder Cirilo[1], Carlos J.P. de Lucena[1], Jan Sudeikat[2], Christian Hahn[3], and Jorge J. Gomez-Sanz[4]

[1] PUC-Rio, Computer Science Department, LES - Rio de Janeiro, Brazil
{ionunes,ecirilo,lucena}@inf.puc-rio.br
[2] Multimedia Systems Laboratory, Hamburg University of Applied Sciences, Berliner Tor 7, 20099 Hamburg, Germany
jan.sudeikat@haw-hamburg.de
[3] German Research Center for Artificial Intelligence (DFKI), Stuhlsatzenhausweg 3, 66123 Saarbrücken
Christian.Hahn@dfki.de
[4] Facultad de Informática, Universidad Complutense de Madrid, Spain
jjgomez@fdi.ucm.es

Abstract. Agent literature has shown a big concern on the production of multi-agent system specifications. Nevertheless, there has not been a similar effort in surveying the different ways an agent oriented specification can be transformed into the actual implementation. This survey intends to cover this gap by pointing at concrete representative works of different implementation approaches.

Keywords: Implementation, agent oriented specification, coding, transformation.

1 Introduction

The survey deals with different existing approaches considered in the translation from a Multi-Agent System (MAS) specification into code and their derived problems, namely, obtaining code from the specification and the consistency between both. As in the general field of software engineering, this remains a major challenge and has motivated research lines where assisted or automated solutions were devised.

Assisted solutions propose team work discipline (processes, guidelines) plus support tools controlled by developers. Automatic solutions provide automatic means to convert the specification into code and, ideally, backwards, leading to what is known as roundtrip engineering.

In both assisted and automatic approaches, developers have to deal with the gap between the concepts used in the specification and those the target platform implements. When the gap is small, the transit from specification to code is smooth and straightforward. Nevertheless, this is not the general case and developers need to find correspondences between the conceptual structures in the

M.-P. Gleizes and J.J. Gomez-Sanz (Eds.): AOSE 2009, LNCS 6038, pp. 169–179, 2011.

specification and those supported by the target platform. This mapping may require as well developing new structures over the target platform to which the translation is more feasible.

To account for the progress made in the specification to code transition, this work considers a selection of recent works, categorising them into assisted transition and automated transition approaches. The gap between specification and code is briefly reviewed in section 2. Assisted translation works are presented in section 3, while automatic translation is regarded in section 4. Finally, we present conclusions in section 5.

2 Dealing with the Gap from Specification to Code

The implementation of a MAS specification is not straightforward most of the times. This is mainly due to the difference between the abstractions provided by the adopted modelling language used in the specification and the ones provided by the target implementation platform (we will assume the programming language is determined by the chosen target platform). When transition is not straightforward, the way to bridge the gap consists of creating mappings from networks of concepts in the specification to code structures in the target platform (it may be existing code structures or new ad-hoc ones). Hence, it is necessary to discuss first what kind of structures can be found at both sides of the gap.

Recognising recurrent structures in MAS specifications and using pre-defined solutions to implement them allows to systematize the process of implementing a MAS. Like in conventional software engineering, this leads to the concept of re-usability, since similar solutions can be applied to similar problems. Software reuse provides benefits, such as increased productivity and quality, for the software development; however there has been little research effort directed toward adopting reuse techniques in the development of MASs [Gir02].

Intuitively, in a reusable solution, there has to be pieces of specifications that map to pieces of code. In this survey, we focus on three ways of managing these associations: patterns [CSC03], modularisation [BHRH00], and aspects [GL08].

Patterns refer to templates of elements to be found in a specification and their corresponding code. In MAS, developers have to deal first with the diversity of agent programming concepts, which interferes with the definition of reusable pattern across programming languages and implementation platforms. Hence, patterns have to be abstract to ensure reuse across agent platforms and programming languages, what prevents, at the same time, to have a stable implementation reference. Nevertheless, establishing a collection of agreed patterns may change this in this future and will force target platforms to tell how these patterns can be implemented. An example of this ideal are FIPA protocols and their standard implementations. FIPA defines semantic and notation for a set of reusable protocols and, for them, one can find FIPA-compliant implementations, like JADE. Hence, in a specification, a developer can use an existing FIPA protocol and find straightforward implementations. Nevertheless, this is not the general case. There are authors working on the identification of meaningful

structures at the specification level, like inter-agent coordination schemes [GVO07] (expressed using node-edge diagrams) [DWK01] (using UML notation) [SR09] (using domain specific languages), or organisational structures [KGM02] (expressed using Tropos). More coherent with the classic notion of pattern is the work [CSC03], where patterns are treated as pieces of specification concepts and their corresponding code, what permits a more adequate reuse across developments. The management of patterns in a specification (finding occurrences, for instance, of patterns collected in a library), may be supported by tools implementing different kinds of transformations, like those considered in section 4.

The identification of reusable structures, i.e. patterns, is closely related to modularisation and componentisation. Agent modules enable the provision of reusable clusters of functionalities. Therefore, they allow to encapsulate and reuse pattern realisations within a development environment as proposals for modularisation concepts concern specific agent architectures. For BDI-like agents, *Capabilities*, which contain reusable sets of beliefs, goals and plans, have been proposed. The works [BHRH00] and in [BP07] show how interactive abilities, e.g. the participation in a contract-net protocol, can be encapsulated and reused by a goal-oriented interface. This module concept is available in the Jack and Jadex agent platforms. Components are more oriented towards the engineering of a system and refer to a reusable coding element whose dependencies and internal behaviour are well defined using some specification language: a component representation makes explicit what is required to make a component work. Agent literature has researched reusable artefacts with different purposes: organisation artefacts [HBKR09] (which are implemented with CArtAgO infrastructure [RVO06]), environments [RVO07] (implemented with CArtAgO) [VHR+07] (review of other infrastructures), coordination in general [RV05] (implemented as TuCSON coordination infrastructure), or facilitating cognitive processes by representing some relevant aspect of the environment [PR08] (implemented with CArtAgO and Jason).

A third possibility is using aspects. Unlike the other solutions, aspects act as a crosscutting concern affecting different, possibly separated, architectural components. This suits well some specification concepts that affect several elements in the implementation. The use of aspect oriented programming for developing MASs brings two main benefits [GL08]: modular reasoning at the architectural level and smooth transition in software life-cycle phases. The aspectization of agent architectures supports modular reasoning, since the architects are able to more independently treat each agent-hood property. Aspect oriented agent architectures are directly mapped to implementation abstractions using well known aspect oriented programming languages, such as AspectJ.

3 Assisted Translation

Assisted translation is about having one or many developers translating a MAS specification into instructions of a programming language using guidelines or programs that support the process. The existence of correspondences between specification structures and code structures may reduce the effort of the MAS

implementation. The main idea is to identify patterns, modules, or aspects in design models, and to implement them using pre-defined solutions, which may be available as reusable code assets in a repository.

In addition, methods and guidelines may help the developer to systematize the coding process. In Moraitis and Spanoudakis [MS06], it is demonstrated how systems designed following the GAIA methodology, and its corresponding models, can be converted to JADE for deployment. The *Gaia2Jade Process* proposes that the implementation phase should be performed in four stages: communication protocol definition, activities refinement, JADE behaviour creation, and agent classes construction.

The Agent-Object-Relationship Modelling Language (AORML) [Wag03] is another proposal for modelling MAS that is mainly inspired by the Agent Oriented Programming proposal of Shoham [Sho93]. A conceptual mapping between AORML and JADE is presented in Taveter and Wagner [TW08]. This mapping should be used by developers as guideline for obtaining the actual code. The same occurs with the mapping between Tropos and AORML as discussed in Guizzardi-Silva et al. [SPD03].

Role modelling [Ken99] has counted as well with the use of guidelines and programming language specific tools. It proposes implementation of agents playing roles using object oriented patterns, or conceiving different roles as aspects associated with the agent.

4 Automatic Translation

In the automatic translation approach, a model is processed to obtain a software system with different degrees of operativility. Automatic translation is generally performed by means of model transformation tools or generative programming approach, but always assuming the existence of a modelling language.

Code generation, whether it comes from a transformation or a generative approach, has been addressed early in the world of agents, back to Zeus methodology [NNLC99] in 1999. Trying to focus on more recent works, a selection of research results have been selected and classified into two broad categories: with and without an agent oriented methodology background.

When there is an agent oriented methodology, there is a modelling language whose use is guided by a method with the aid of development tools. Some of these tools are intended to assist the creation of the specification while others to process the specification, for instance, to generate a system. These tools are used with different intentions depending on the development stage according to the kind of products the development process demands. For instance, a developer can expect during the implementation stage to have code generation facilities producing the actual code of the system, while at the testing stage, the developer wants executable code that tests the automatically generated system.

Other works start directly from an agent modelling language without a special focus on the method. Hence, there is little concern about development stages and a simple design-generate-code approach is followed. This simplifies the construction of a code generation approach, but leaves to the developer the burden

of deciding how and when to apply it. In a way, this kind of works can be seen as a germen for a future methodology.

Despite the focus, it can be appreciated that automatic translation of specification to code is strongly associated with model driven support tools. Model Driven Development (MDD) focuses on the construction of a model of the system and how to transform this model into source code, not just binaries. This is the case of the Eclipse Modelling Framework, which enables the construction of meta-models and, from them, build visual editors and generate code. Within this category fall the evolution of PDT tool [STP10], SDLMAS [CSZ09], O-MaSE [GODR09], TROPOS [BDM+06], and DSML4MAS Development Environment [WH09]. Other works prefer to reuse UML modelling tools, like the Agent Modelling Language (AML) [CTCG04, ICG06], PASSI [Cos05], and MAS-ML [dMdSLC05]. From these works, only INGENIAS [PGSF06] proposes custom facilities based on the meta-modelling framework INGENME (`http://ingenme.sourceforge.net`).

4.1 Agent Oriented Methodologies

In this category, five works are introduced. They are briefly presented in table 1 with special focus on the following characteristics: which one is the underlying development method; the name of the modelling language; the name of the tool that supports the translation and coding; and the target platform. A concrete description of each work follows.

O-MaSE [GODR09] is an evolution of MaSE methodology that incorporates the concept of Organisation. This evolution is supported by the editor agentTool III. This tool permits to create the specification of the MAS and, at the same time, transform it into executable code. The target platform is the JADE agent platform.

Pavón et al. [PGSF06] presented an update to INGENIAS introducing the *INGENIAS Development Kit (IDK)*, as a way to provide Model Driven Development tools for MAS development. It presents the IDK MAS Model Editor, a graphical tool for MAS model creation, and a modular approach adapt the editor and tools to new metamodels or target platforms. It also proposes that the model generation and metamodel development should be performed in parallel with periodic consistency checks to allow feedback from one activity to the other during the development.

Prometheus methodology permits to generate skeleton code for Jack agent platform in the Eclipse based implementation of the PDT tool [STP10]. This tool provides a equivalent representation of PDT concepts using Eclipse facilities plus menu options to transform specifications into Jack agent programs.

PASSI [Cos05] proposes a step-by-step refinement from requirements to code. The generation of the system is performed in part by the PASSI Toolkit (PTK). The other part of code generation is responsibility of AgentFactory tool. It regards code reuse in form of predefined patterns of agents and tasks. PTK compiles PASSI diagrams, exports the specification to AgentFactory or produces skeleton for current agents. AgentFactory works at the pattern level by letting

Table 1. Reviewed automatic code generation approaches from agent oriented methodologies

Method	Modeling Language	Tool	Target Platform
O-MaSE	Based on O-MaSE's metamodel /AUML Interaction Diagrams	agentTool III	JADE platform [GODR09]
INGENIAS	Based on INGENIAS metamodel	INGENIAS Development Kit (IDK)	JADE platform [GSFPGM08]
Prometheus	Prometheus	Prometheus Design Tool (PDT)	JACK [STP10]
Tropos	Based on i*	JADE Template Generator	JADE [BDM+06]
PASSI	UML	Rational Rose UML CASE tool + add-in supporting PASSI (PTK) + AgentFactory	JADE and FIPA-OS [Cos05]

the developer choose among a set of predefined designs for which there is actual code.

Bertolini et al. [BDM+06] describe how their agent-based software development process can be performed through MDD. For this purpose, the authors demonstrated how automatic transformations can be used to convert UML models on the various phases of the TROPOS methodology to finally maintain the related implementation. The model transformation is defined by using Tefkat model transformation engine.

4.2 Agent Oriented Modeling Languages

This section includes four works that mainly focus on the modeling language. Their features are summarized in table 2. This table addresses similar characteristics as in the previous section without regarding the development method. Details of each work follow.

Agent-DSL [KGL04] is an aspect oriented modeling language that permits to model agent features like knowledge, autonomy, adaptation, or roles. An agent oriented specification using Agent-DSL is converted into code using an aspect oriented agent architecture. Code generation uses Eclipse facilities to combine the XML containing the Agent-DSL specification with some templates.

The Agent Modelling Language (AML) [CTCG04, ICG06] was designed to address the specific qualities offered by MAS that are difficult to model with object oriented modelling languages like UML. The authors of [CGT06] discuss an automatic transformation that is based on a CASE modelling tool to define code generators that translate the AML specification into code provided by the Living Systems Technology Suite [RGK06], which is a development environment for

Table 2. Reviewed automatic code generation approaches for modeling languages

Modeling Language	Tool	Target Platform
Agent-DSL	Agent AO-Architecture Eclipse Plug-in	- (Java and AspectJ) [KGL04]
MAS-ML	VisualAgent	ASF Framework [dMdSLC05]
SDLMAS	Eclipse SDLMAS plug-in	JADE platform [CSZ09]
DSML4MAS	DSML4MAS Development Environment	JACK and JADE [Hah08]

producing applications based on software agent technology. The Living Systems Technology Suite (LS/TS) provides a Java-based foundation for the development and operation of products and solutions based on software agent technology.

MAS-ML [dMdSLC05] is a modelling language, which extends the UML metamodel to include agent concepts. MAS-ML specification can be transformed into code addressing the ASF framework. The process is performed using VisualAgents environment. This tool has a graphical facilities for modelling; a transformation tool from the MAS-ML to UML XMI; and a code generation tool.

SDLMAS [CSZ09] is oriented towards the modeling of complex interactions. The description of interactions is made through scenarios. Scenario description is made using a text file, which follows the SDLMAS language conventions. These files contains as well agent definitions, agent actions, and roles. They are processed to produce code compatible with the SDLMAS platform, which is built over the JADE platform. Code generation reuses Eclipse facilities to convert the SDLMAS text file into EMF data and then to transform the EMF representation into code.

In [Hah08], a model transformation in accordance to MDD between a metamodel called PIM4Agents and the agent platforms JACK and JADE has been discussed. Moreover, a formal semantics has been defined as well as a model-driven methodology. The DSML4MAS Development Environment [WH09], a graphical editor based on the Eclipe's Graphical Modelling Framework, provides a clear concrete syntax to design MAS in accordance to DSML4MAS.

5 Conclusions

This work has reviewed a number of works in recent years of research in MAS. It surveys different ways the agent research community is addressing the specification to code transition problem. Works have been divided into two broad categories (assisted and automatic) while some theoretical principles underlying both approaches (structures to be mapped between the specification and the design) were introduced.

Readers should not assume this survey considers in detail the implementation stage. The discussed works mainly concern the translation of agent specifications to source code. Nevertheless, there are issues that bias an implementation that have not been regarded. From a single specification, different implementations are possible, but only a few will satisfy the actual requirements of a development. The reason is that there are requirements that refer to non-functional issues, such as security, resource usage, performance, or scalability. Finding appropriate code structures that satisfy them adds more complexity to the specification-to-code translation problem. Hence, they deserve attention from the developer, as well.

Acknowledgements

Jorge J. Gomez-Sanz has partially been funded by the the project Agent-based Modelling and Simulation of Complex Social Systems (SiCoSSys), supported by Spanish Council for Science and Innovation, with grant TIN2008-06464-C03-01, and by the Programa de Creación y Consolidación de Grupos de Investigación UCM-Banco Santander for the group number 921354 (GRASIA group). Ingrid Nunes #141278/2009-9, Elder Cirilo #142013/2008-0, Carlos Lucena #304810/2009-6 thank CNPq for respective research grants.

References

[BDM+06] Bertolini, D., Delpero, L., Mylopoulos, J., Novikau, A., Orler, A., Penserini, L., Perini, A., Susi, A., Tomasi, B.: A Tropos model-driven development environment. In: Proceedings of the 18th Conference on Advanced Information Systems Engineering (CAiSE 2006), Forum Proceedings, Theme: Trusted Information Systems, Luxembourg, June 5-9. CEUR Workshop Proceedings, vol. 231, CEUR-WS.org (2006)

[BHRH00] Busetta, P., Howden, N., Rönnquist, R., Hodgson, A.: Structuring BDI agents in functional clusters. In: Jennings, N.R. (ed.) ATAL 1999. LNCS, vol. 1757, pp. 277–289. Springer, Heidelberg (2000)

[BP07] Braubach, L., Pokahr, A.: Goal-oriented interaction protocols. In: Petta, P., Müller, J.P., Klusch, M., Georgeff, M. (eds.) MATES 2007. LNCS (LNAI), vol. 4687, pp. 85–97. Springer, Heidelberg (2007)

[CGT06] Cervenka, R., Greenwood, D., Trencansky, I.: The AML approach to modeling autonomic systems. In: Proceedings of the International Conference on Autonomic and Autonomous Systems (ICAS 2006), p. 29. IEEE Computer Society, Los Alamitos (2006)

[Cos05] Cossentino, M.: From Requirements to Code with the PASSI Methodology, ch. IV, pp. 89–106. Idea Group Inc., Hershey (2005)

[CSC03] Cossentino, M., Sabatucci, L., Chella, A.: Patterns reuse in the passi methodology. In: Zhang, S.-W., Petta, P., Pitt, J. (eds.) ESAW 2003. LNCS (LNAI), vol. 3071, pp. 294–310. Springer, Heidelberg (2004)

[CSZ09] Cavrak, I., Stranjak, A., Zagar, M.: Sdlmas: A scenario modeling framework for multi-agent systems. Journal of Universal Computer Science (JUC.S) 15(4), 898–925 (2009)

[CTCG04] Cervenka, R., Trencanský, I., Calisti, M., Greenwood, D.A.P.: AML: Agent modeling language toward industry-grade agent-based modeling. In: Odell, J.J., Giorgini, P., Müller, J.P. (eds.) AOSE 2004. LNCS, vol. 3382, pp. 31–46. Springer, Heidelberg (2005)

[dMdSLC05] de Maria, B.A., da Silva, V.T., Lucena, C., Choren, R.: Visualagent: A software development environment for multi-agent systems. In: Proceedings of the 19th Brazilian Symposiun on Software Engeneering (SBES 2005), Tool Track (2005)

[DWK01] Deugo, D., Weiss, M., Kendall, E.: Reusable patterns for agent coordination, pp. 347–368 (2001)

[Gir02] Girardi, R.: Reuse in agent-based application development. In: First International Workshop on Software Engineering for Large-Scale Multi-Agent Systems, SELMAS 2002 (2002)

[GL08] Garcia, A., Lucena, C.: Taming heterogeneous agent architectures. Commun. ACM 51(5), 75–81 (2008)

[GODR09] Garcia-Ojeda, J.C., DeLoach, S.A., Robby: agenttool iii: from process definition to code generation. In: AAMAS 2009: Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems, pp. 1393–1394. International Foundation for Autonomous Agents and Multiagent Systems (2009)

[GSFPGM08] Gomez-Sanz, J.J., Fuentes, R., Pavón, J., Ivan, G.-M.: Ingenias development kit: a visual multi-agent system development environment. In: AAMAS 2008: Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems, pp. 1675–1676. International Foundation for Autonomous Agents and Multiagent Systems (2008)

[GVO07] Gardelli, L., Viroli, M., Omicini, A.: Design patterns for self-organizing systems. In: Burkhard, H.-D., Lindemann, G., Verbrugge, R., Varga, L.Z. (eds.) CEEMAS 2007. LNCS (LNAI), vol. 4696, pp. 123–132. Springer, Heidelberg (2007)

[Hah08] Hahn, C.: A domain specific modeling language for multiagent systems. In: Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2008), Estoril, Portugal, May 12-16, vol. 1, pp. 233–240. IFAAMAS (2008)

[HBKR09] Hübner, J.F., Boissier, O., Kitio, R., Ricci, A.: Instrumenting multiagent organisations with organisational artifacts and agents. In: Autonomous Agents and Multi-Agent Systems (2009)

[ICG06] Ivan, T., Cervenka, R., Greenwood, D.: Applying a uml-based agent modeling language to the autonomic computing domain. In: Conference on Object Oriented Programming Systems Languages and Applications, pp. 521–529. ACM Press, New York (2006)

[Ken99] Kendall, E.A.: Role modeling for agent system analysis, design, and implementation. In: ASA/MA, pp. 204–218. IEEE Computer Society, Los Alamitos (1999)

[KGL04] Kulesza, U., Garcia, A., Lucena, C.: Generating aspect-oriented agent architectures. In: Proceedings of the 3rd Workshop on Early Aspects - Aspect-Oriented Requirements Engineering and Architecture Design, 3rd International Conference on Aspect-Oriented Software Development (AOSD 2004), Lancaster, UK (2004)

[KGM02] Kolp, M., Giorgini, P., Mylopoulos, J.: A goal-based organizational perspective on multi-agent architectures. In: Meyer, J.-J.C., Tambe, M. (eds.) ATAL 2001. LNCS (LNAI), vol. 2333, pp. 128–140. Springer, Heidelberg (2002)

[MS06] Moraitis, P., Spanoudakis, N.I.: The Gaia2Jade Process for Multi-Agent Systems Development.. Applied Artificial Intelligence 20(2-4), 251–273 (2006)

[NNLC99] Nwana, H.S., Ndumu, D.T., Lee, L.C., Collis, J.C.: Zeus: A toolkit for building distributed multiagent systems. Applied Artificial Intelligence 13(1-2), 129–185 (1999)

[PGSF06] Pavón, J., Gómez-Sanz, J.J., Fuentes, R.: Model Driven Development of Multi-Agent Systems.. In: ECMDA-FA, pp. 284–298 (2006)

[PR08] Piunti, M., Ricci, A.: Cognitive use of artifacts: Exploiting relevant information residing in mas environments. In: Meyer, J.-J.C., Broersen, J. (eds.) KRAMAS 2008. LNCS, vol. 5605, pp. 114–129. Springer, Heidelberg (2009)

[RGK06] Rimassa, G., Greenwood, D., Kernland, M.E.: The Living Systems Technology Suite: An autonomous middleware for autonomic computing. In: Proceedings of the International Conference on Autonomic and Autonomous Systems (ICAS 2006), Washington, DC, USA, July 16-21, p. 33. IEEE Computer Society, Los Alamitos (2006)

[RV05] Ricci, A., Viroli, M.: Coordination artifacts: A unifying abstraction for engineering environment-mediated coordination in mas. Informatica 29, 433–443 (2005)

[RVO06] Ricci, A., Viroli, M., Zhang, S.-W.: cArtAgO: A framework for prototyping artifact-based environments in MAS. In: Weyns, D., Van Dyke Parunak, H., Michel, F. (eds.) E4MAS 2006. LNCS (LNAI), vol. 4389, pp. 67–86. Springer, Heidelberg (2007)

[RVO07] Ricci, A., Viroli, M., Omicini, A.: Give agents their artifacts: the a&a approach for engineering working environments in mas. In: AAMAS, p. 150. IFAAMAS (2007)

[Sho93] Shoham, Y.: Agent-oriented programming. Artificial Intelligence 60(1), 51–92 (1993)

[SPD03] Guizzardi-Silva Souza, R., Perini, A., Dignum, V.: Using intentional analysis to model knowledge management requirements in communities of practice. Technical Report TR-CTIT-03-53, Centre for Telematics and Information Technology, University of Twente, Enschede (2003) ISSN 1381-3625

[SR09] Sudeikat, J., Renz, W.: MASDynamics: Toward systemic modeling of decentralized agent coordination. In: Kommunikation in Verteilten Systemen. Informatik aktuell, pp. 79–90 (2009)

[STP10] Sun, H., Thangarajah, J., Padgham, L.: Eclipse-based prometheus design tool. In: van der Hoek, W., Kaminka, G.A., Lespérance, Y., Luck, M., Sen, S. (eds.) AAMAS, pp. 1769–1770. IFAAMAS (2010)

[TW08] Taveter, K., Wagner, G.: Agent-oriented modeling and simulation of distributed manufacturing. In: Handbook of Research on Nature-Inspired Computing for Economics and Management, pp. 527–540. Idea Group Reference (2008)

[VHR+07] Viroli, M., Holvoet, T., Ricci, A., Schelfthout, K., Zambonelli, F.: Infrastructures for the environment of multiagent systems. Autonomous Agents and Multi-Agent Systems 14(1), 49–60 (2007)

[Wag03] Wagner, G.: The Agent-Object-Relationship meta-model: Towards a unified view of state and behavior. Information Systems 28(5), 475–504 (2003)

[WH09] Warwas, S., Hahn, C.: The dsml4mas development environment. In: AAMAS 2009: Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems, Richland, SC, pp. 1379–1380. International Foundation for Autonomous Agents and Multiagent Systems (2009)

Testing in Multi-Agent Systems

Cu D. Nguyen[1], Anna Perini[1], Carole Bernon[2],
Juan Pavón[3], and John Thangarajah[4]

[1] Center for Information Technology FBK-IRST, Trento, Italy
{cunduy,perini}@fbk.eu
[2] IRIT, University of Toulouse, Toulouse, France
bernon@irit.fr
[3] GRASIA, Universidad Complutense de Madrid, Madrid, Spain
jpavon@fdi.ucm.es
[4] RMIT University, Melbourne, Australia
john.thangarajah@rmit.edu.au

Abstract. Testing software agents and Multi-Agent Systems (MAS) needs suitable techniques to evaluate agent's autonomous behaviours as well as distribution, social and deliberative properties, which are particular to these systems. This paper reviews testing methods and techniques with respect to the MAS properties they are able to address. For this purpose, we provide a reference framework that provides a classification of MAS testing levels (such as unit, agent, integration, system, and acceptance) and of testing approaches along the development artefact they exploit (namely, design and code artefacts). Open issues in testing MAS are then discussed providing a basis for a research roadmap.

1 Introduction

Software testing aims at answering questions like *Is the system being built good enough? Does the resulting system fulfil the requirements?* That is, testing is the software development activity, devoted to evaluating product quality and improving it by identifying defects and problems.

The specific nature of software agents, which are designed to be distributed, autonomous, and deliberative makes it difficult to apply existing software testing techniques to them. For instance, agents operate asynchronously and in parallel, which challenges testing and debugging. Agents communicate primarily through message passing instead of method invocation, so traditional testing approaches are not directly applicable. Agents are autonomous and cooperate with other agents, so they may run correctly by themselves but incorrectly in a community or vice-versa. Moreover, agents can be programmed to learn (i.e., change their behaviour); so successive tests with the same test data may give different results. Initial works on evaluating MAS quality focused on the definition of techniques for automating the validation of MAS specifications through formal proofing or model-checking [2, 20], and on the development of debugging techniques and tools to enhance MAS development platforms [15, 3, 31].

M.-P. Gleizes and J.J. Gomez-Sanz (Eds.): AOSE 2009, LNCS 6038, pp. 180–190, 2011.

Structured testing approaches have been proposed more recently, to complement analysis and design methodologies [4, 26, 31, 14]. These approaches rest on the idea that the behaviour of the MAS can be dynamically evaluated providing as input a set of test cases that are derived from analysis and design artefacts.

Differently from these techniques, simulation-based approaches aim at detecting abnormal behaviours while a simplified version of the system (or a model of it) is executed in a virtual environment [13, 35, 6, 18]. These approaches seem to be particularly appropriate to evaluate emerging behaviours in self-organizing systems [7]. Data mining techniques have been applied to analyse simulation logs for large MAS, as it is the case in ACLAnalyser [33, 34], which is a way to cope with scalability of MAS.

In this paper we survey testing methods and techniques, characterizing them along the testing levels they support (such as unit, agent, integration, system, and acceptance). We differentiate them also between simulation-based techniques (called *passive* approaches) and structured testing methodologies (called *active* approaches).

The paper is structured as follows. Section 2 presents a MAS testing framework. State of the art MAS testing approaches are then surveyed in Section 3. Open issues in testing MAS and promising testing techniques are then discussed in Section 4, providing a basis for a research roadmap.

2 Classification Dimensions

In order to provide the reader with a structured view for clarity, we propose to classify existing work on MAS testing following two different aspects: *testing levels* and *testing techniques*. Testing in MAS consists of five levels, as proposed in [23] and [22]: *unit, agent, integration, system*, and *acceptance*. The testing objectives, subjects to test, and activities of each level are described as follows:

- *Unit*. Test all units that make up an agent, including blocks of code, implementation of agent units like goals, plans, knowledge base, reasoning engine, rules specification, and so forth; make sure that they work as designed.
- *Agent*. Test the integration of the different modules inside an agent; test agents' capabilities to fulfil their goals and to sense and effect the environment.
- *Integration* or *Group*. Test the interaction of agents, communication protocol and semantics, interaction of agents with the environment, integration of agents with shared resources, regulations enforcement; Observe emergent properties, collective behaviours; make sure that a group of agents and environmental resources work correctly together.
- *System* or *Society*. Test the MAS as a system running at the target operating environment; test the expected emergent and macroscopic properties of the system as a whole; test the quality properties that the intended system must reach, such as adaptation, openness, fault-tolerance, performance.
- *Acceptance*. Test the MAS in the customer's execution environment and verify that it meets stakeholder goals, with the participation of stakeholders.

We will organise existing work along these levels to see on which levels each work focuses[1].

We can also use testing techniques in terms of their test input perspective: *passive* and *active*, to classify existing work. Some employ the passive perspective to solely observe the output behaviours, test inputs are often predefined configurations or ignored; while others adopt the active perspective seeking extensively for good test inputs and at the same time monitoring the output behaviours of the subjects under test.

It is worthwhile noting that the two chosen dimensions, i.e. testing levels and test techniques, are not intended to be comprehensive. They rather provide a useful framework to systematically organise existing work. The next section will give a survey of the most relevant and active work on MAS testing according to these classification dimensions.

Moreover, the work surveyed are also assessed in terms of their maturity into *usable*, *in progress*, and *concept* as follows: *Usable:* there are published results, there is available material that permits to reproduce the results, and the work has been used in the development of medium or big size projects. *In progress:* there are published results, there is available material that permits to reproduce the results. So far, the work has been used in academic size projects. *Concept:* there are published results, but the applicability of the work is still to be proven.

3 MAS Testing Approaches

The data in Table 1 give an overview of the most recent and active work in this domain. They are organized according to the classification introduced in the previous section. The third dimension, *maturity*, will be tagged to each work. In the following sections, we first summarize the contributions of the works that focus mainly on a particular testing level, e.g. *agent*; hereafter are the works that tackle more than one testing level. In addition, we also summarize some interesting work that do early validation of the agent and MAS design based on simulation techniques and generated agent skeletons. It is non-trivial to organize

Table 1. State-of-the-art work on MAS testing

	Unit	***Agent***	***Integration***	***System***	***Acceptance***
Active	Zhang et al. (2007, 2008, 2009), Tiryaki et al. (2006), Ekinci et al. (2008) , Nguyen et al. (2010)	Núñez et al. (2005), Coelho et al. (2006), Tiryaki et al. (2006), Gómez-Sanz et al. (2009), Nguyen et al. (2008, 2009, 2010)	Gómez-Sanz et al. (2009), Nguyen et al. (2010)	Nguyen et al. (2010)	Nguyen et al. (2010)
Passive		Lam and Barber (2005), Núñez et al. (2005), Gardelli et al. (2005), Fortino et al. (2006), Bernon et al. (2007), Cossentino et al. (2008)	Sierra et al. (2004), Botía et al. (2004), Rodrigues et al. (2005), Serrano and Bota (2009), Serrano et al. (2009), Sudeikat and Renz (2009)	De Wolf et al. (2005)	

[1] These works are organized in chronological order; those published in the same year are ordered alphabetically.

these works following the proposed classification since their subjects under test are agent design, not agent code. However, we decide to include them because they are complementary to testing, and both types contribute to the final goal of ensuring the quality of the agent or MAS under development.

3.1 Unit Level

Zhang et al. [38, 39, 40], *in progress*: This body of work presents a framework for automated unit testing of agent systems. The approach is a model-based testing approach where the models used are the design artefacts of the agent design methodology. The chosen methodology (though it is extensible to other methodologies with similar concepts) is Prometheus [28] and the corresponding tool that incorporates the testing techniques is the Prometheus Design Tool (PDT[2]) [39].

In [38], the basic units for testing are identified as events, plans and beliefs. Events are tested for coverage (does this event get handled) and overlap (are there multiple plans that can handle this event in the same situation). Plans are tested for whether the plan gets triggered in at least some situations (but not all if there is a condition specified to its applicability), does the plan complete[3], and does the plan post the events that it should (as indicated in the design). The testing of beliefs are limited to verifying whether the data fields as indicated in the design are implemented and if there are any events posted by the belief (e.g., if an event is posted when a belief is updated), test if these events do get posted.

The paper details mechanisms for identifying the order in which the units are to be tested, for example, if unit x depends on unit y then y must be tested before x. It also details the process of generating test cases and outlines the overall testing process. All the mechanisms are fully automated, but allow for user input at various stages.

In [40] the details of how variables are to be specified, extracted and assigned values to create test cases are presented. Furthermore, in order to execute the test cases the system environment needs to be setup. For example, there may be interaction with an external program within a plan unit (e.g. a query to an external database of a web service). There may also be interaction with another agent within a testing unit. This work deals with the above by introducing initialization procedures and mock agents to simulate other agents/systems.

Ekinci et al. [10], *in progress*: This work considers agent goals as the smallest testable units in MAS and proposes to test these units by means of *test goals*. Each test goal is conceptually decomposed into three sub-goals: *setup*, *goal under test*, and *assert*. The first and last goals prepare pre-conditions and check post-conditions respectively, while testing the goal under test.

[2] www.cs.rmit.edu.au/agents/pdt

[3] Note, here it tests for plan completion not for plan success.

3.2 Agent Level

Lam and Barber [19], *in progress*: This work proposes a semi-automated process for comprehending software agent behaviours. The approach imitates what a human user, can be a tester, does in software comprehension: building and refining a knowledge base about the behaviours of agents, and using it to verify and explain behaviours of agents at runtime.

Núñez et al. [27], *in progress*: This paper introduces a formal framework to specify the behaviour of autonomous e-commerce agents. The desired behaviours of the agents under test are presented by means of a new formalism, called *utility state machine* that embodies users' preferences in its states. Two testing methodologies are proposed to check whether an implementation of a specified agent behaves as expected (i.e., conformance testing); one active, the other passive. In their *active* testing approach, they use for each agent under test a *test* (a special agent) that takes the formal specification of the agent to facilitate it to reach a specific state. The operational trace of the agent is then compared to the specification in order to detect faults. On the other hand, the authors also propose to use *passive* testing, in which the agents under test are observed only, not simulated like in active testing. Invalid traces, if any, are then identified thanks to the formal specifications of the agents.

Coelho et al. [5], *in progress*: Inspired by JUnit [12], this paper proposes a framework for unit testing of MAS based on the use of *Mock Agents*. Their work focuses on testing roles of agents. Mock agents that simulate real agents in communicating with the agent under test were implemented manually; each corresponds to one agent role.

Nguyen et al. [25], *in progress*: This work takes advantage of agent interaction ontologies that define the semantics of agent interactions to: (i) generate test inputs; (ii) guide the exploration of the input space during generation; and, (iii) verify messages exchanged among agents with respect to the defined interaction ontology. A set of generation rules have been defined and using them an automated testing framework can test a particular agent extensively through a large and diverse number of test cases.

Nguyen et al. [24], *in progress*: Agent autonomy makes testing harder: autonomous agents may react in different ways to the same inputs over time, because, for instance they have changeable goals and knowledge. Testing of autonomous agents requires a procedure that caters for a wide range of test case contexts, and that can search for the most demanding of these test cases. Nguyen et al. [24] introduce and evaluate an approach to testing autonomous agents that uses evolutionary optimization to generate demanding test cases. In this work, the authors have proposed a systematic way of evaluating the quality of autonomous agents. First, stakeholder requirements are represented as quality measures, and corresponding thresholds are used as testing criteria. Autonomous agents need to meet these criteria in order to be reliable. Fitness functions that

represent testing objectives are defined accordingly, and guide this evolutionary test generation technique to generate test cases automatically.

3.3 Integration Level

Serrano and Botía [33], Serrano et al. [34], *usable*: One of the issues when testing MAS is their scalability, because of the number of agents and more specifically the huge number of interactions that may arise. This makes it very difficult to apply classical testing techniques. ACLAnalyser addresses these issues by applying data mining techniques on the logs of MAS executions (initially, on the JADE agent platform). This allows discovering emergence patterns at system (social) level. For instance, the creation of communities of agents with strong interaction links can be detected with the clustering facilities of ACLAnalyser.

3.4 Multiple Testing Levels

Tiryaki et al. [37], *in progress*: This work proposes a test-driven MAS development approach that supports iterative and incremental MAS construction. A testing framework called SUnit, which was built on top of JUnit[12] and Seagent [9], was developed to support the approach. The framework allows writing tests for agent behaviours and interactions between agents.

Gómez-Sanz et al. [14], *usable*: This work focuses on agent and interaction testing level. The INGENIAS Development Kit also provides facilities for MAS testing. It relies on the model-driven approach of INGENIAS, which allows the specification of test suites when modelling the MAS. It is possible to specify testing deployments, interaction tests, and agent mental state inspection. MAS models are then transformed into code on the INGENIAS Agent Framework (running on top of JADE agent platform), where tests can be executed and the developer can get information on the progression of organization workflows, interactions, and agents' mental states. Test suites can be automated, and this facilitates model-driven agile development, as it is easy to perform iterations, each one passing a test suite.

Nguyen et al. [26], *in progress*: The paper proposes a comprehensive testing methodology, called Goal-Oriented Software Testing (GOST), that complements and exploits goal-oriented analysis and design to derive test suites at all testing levels, from unit to acceptance. GOST provides a testing process model that brings the connections between goals and test cases explicit and a systematic way of deriving test cases from goal analysis. This can help discovering problems early, avoiding implementing erroneous specifications. In addition, GOST comes with a tool that supports test case generation, specification, and execution.

3.5 Simulation-Based Design Validation

Sierra et al. [35], *usable*: The IDE-eli (Integrated Development Environment for Electronic Institutions) simulation tools support the engineering of MAS as

electronic institutions. The SIMDEI tool, based on Repast, enables to animate and analyze specifications before deploying them, therefore facilitating the location of unexpected behaviours that may jeopardize critical applications. An agent skeleton is automatically generated from its specification, depending on the roles and interactions in which this agent may participate. This skeleton has to be completed by designers with decision-making mechanisms. The simulation process involves different populations of agents with different features that are able to act in a specified institution. The institution designer analyses results and may return to the design stage if results differ from the expected ones.

De Wolf et al.[8], *usable*: This paper proposes an empirical analysis approach combining agent-based simulations and numerical algorithms for analyzing the global behaviour of a self-organizing system. The initial values of macroscopic variables (those related to the properties that are studied) are supplied, a certain number of simulations are then initialized accordingly, simulations are then executed for a predefined duration, the average values of these variables on all the simulations are then given as results to the analysis algorithm, this latter processes these results to compute the next initial values. The algorithm also decides on the configuration of the next simulations (initial conditions, durations) and the cycle is repeated until the algorithm reaches its goal (for instance, converge towards a value, find a steady state).

Gardelli et al. [13], *usable*: Detection of agents having abnormal behaviours in a self-organizing/open MAS. The infrastructure is both based on TuCSoN (agents and artefacts) and principles coming from the human immune system. The behaviour of different scenarios is simulated using SpiM specifications (PI-calculus).

Fortino et al. [11], **Cossentino et al.** [6], *usable*: A simulation-driven development process is obtained by integrating state-chart-based simulation methodology for MAS with PASSI. Simulation methodology based on three phases: modelling, coding and simulation. Simulation is based on MASSIMO, a Java-based discrete-event simulation framework that enables validation and evaluation of the dynamic behaviours of agents (using execution traces) as well as the performance of the system (using evaluation parameters defined in an ad hoc way).

Bernon et al. [1], *in progress*: This paper proposes a model for a cooperative agent and a simulation tool for helping designers of self-organizing MAS. According to the AMAS (Adaptive MAS) theory, the collective function of these systems emerges from the cooperative interactions between agents. Agents have to always avoid, or detect and then repair situations that are against this cooperative social attitude. The proposed tool enables a designer to build a simplified/prototype of his target AMAS under SeSAM, to run simulations in order to get information about the situations that are non cooperative and not processed in the right way by the cooperative agents. The designer can then observe

the behaviour of these agents in order to verify whether their cooperative attitude is the expected one according to the emergent collective function obtained. He has then the possibility to manually improve behaviours by acting on what and how agents perceive, decide and act.

Sudeikat and Renz [36], *in progress*: The validation procedure consists in making hypotheses on the macroscopic observable MAS behaviour based on the MAS designs. These hypotheses are validated by tailoring simulation settings and analyzing the obtained results. Causal Loop Diagrams are used to express the intended application behaviour. An environment of execution is proposed to measure the correlations between state variables and check the presence of causal relations.

4 Open Issues

The above-described survey allows pointing out open problems, and solutions to some of them that have not been validated empirically, so defining the basis of a research agenda for MAS testing.

First, looking at the five testing levels we may notice that most of the available approaches concern the unit, agent and integration testing levels, leading to the observation that system and acceptance testing need further investigation.

Second, most of the approaches need to be consolidated and evaluated on realistic case studies to provide evidence of their usability.

Specific properties of MAS systems, such as autonomy and adaptivity, and more generally self-* properties[4], are challenging research on engineering and testing such software systems. Similar issues were previously pointed out in the Agentlink's research roadmap [21], quoting it *...techniques are needed to ensure that the system still executes in an acceptable, or safe, manner during the adaptation process, for example using techniques such as dependency analysis or high level contracts and invariants to monitor system correctness before, during and after adaptation.*

While addressing the above mentioned open issues, possible benefits from conventional software engineering testing's techniques are worth considering. For instance, extending to Multi-Agent systems research on metrics for evaluating software qualities and for defining test oracles for systems' properties seem particularly promising. Examples are recent works in the context of the definition of self-organisation and emergence mechanisms for achieving self-* properties, [16] and [32], which study a certain number of metrics for evaluating adaptivity in self-organizing systems. Such metrics could be proved useful when testing and validating self-organising and complex systems. Moreover, Mikhail et al. [30, 29] apply coupling and cohesion metrics to evaluate qualities of service-based systems and intend to apply a similar approach to define metrics for agents within service oriented architectures.

[4] Namely self-managing, self-configuring, self-healing, self-optimizing, and selfprotecting, as defined in the autonomic paradigm [17].

References

[1] Bernon, C., Gleizes, M.P., Picard, G.: Enhancing self-organising emergent systems design with simulation. In: O'Hare, G.M.P., Ricci, A., O'Grady, M.J., Dikenelli, O. (eds.) ESAW 2006. LNCS (LNAI), vol. 4457, pp. 284–299. Springer, Heidelberg (2007)

[2] Brazier, F.M.T., Dunin-Keplicz, B., Jennings, N.R., Treur, J.: DESIRE: Modelling Multi-Agent Systems in a Compositional Formal Framework. Int. J. Cooperative Inf. Syst. 6(1), 67–94 (1997)

[3] Caire, G., Cossentino, M., Negri, A., Poggi, A., Turci, P.: Multiagent Systems Implementation and Testing. In: Proc. of the 4th From Agent Theory to Agent Implementation Symposium, AT2AI-4 (2004)

[4] Chella, A., Cossentino, M., Sabatucci, L., Seidita, V.: Agile passi: An agile process for designing agents. International Journal of Computer Systems Science & Engineering (2006)

[5] Coelho, R., Kulesza, U., von Staa, A., Lucena, C.: Unit testing in multi-agent systems using mock agents and aspects. In: SELMAS 2006: Proceedings of the 2006 International Workshop on Software Engineering for Large-scale Multi-agent Systems, pp. 83–90. ACM Press, New York (2006), http://dx.doi.org/http://doi.acm.org/10.1145/1138063.1138079

[6] Cossentino, M., Fortino, G., Garro, A., Mascillaro, S., Russo, W.: PASSIM: A Simulation-based Process for the Development of Multi-Agent Systems. Int. Journal of Agent-Oriented Software Engineering 2(2), 132–170 (2008)

[7] De Wolf, T.: Analysing and Engineering Self-organising Emergent Applications, PhD thesis, Katholieke Universiteit Leuven (2007)

[8] De Wolf, T., Samaey, G., Holvoet, T.: Engineering self-organising emergent systems with simulation-based scientific analysis. In: Brueckner, S., Serugendo, D.M., Hales, D., Zambonelli, F. (eds.) Third International Workshop on Engineering Self-Organising Application, sUtrech, Netherlands, pp. 146–160 (2005)

[9] Dikenelli, O., Erdur, R.C., Gumus, O.: Seagent: a platform for developing semantic web based multi agent systems. In: AAMAS 2005: Proceedings of the Fourth International Joint Conference on Autonomous Agents and Multiagent Systems, pp. 1271–1272. ACM Press, New York (2005), http://dx.doi.org/http://doi.acm.org/10.1145/1082473.1082728

[10] Ekinci, E.E., Tiryaki, A.M., Cetin, O., Dikenelli, O.: Goal-Oriented Agent Testing Revisited. In: Proc. of the 9th Int. Workshop on Agent-Oriented Software Engineering, pp. 85–96 (2008)

[11] Fortino, G., Garro, A., Russo, W., Caico, R., Cossentino, M., Termine, F.: Simulation-Driven Development of Multi-Agent Systems. In: Workshop on Multi-Agent Systems and Simulation, Palermo, Italia (2006)

[12] Gamma, E., Beck, K.: JUnit: A Regression Testing Framework (2000), http://www.junit.org

[13] Gardelli, L., Viroli, M., Omicini, A.: On the Role of Simulations in the Engineering of Self-Organising MAS: The Case of an Intrusion Detection System in TuCSoN. In: 3rd International Workshop Engineering Self-Organising Applications, pp. 161–175 (2005)

[14] Gómez-Sanz, J.J., Botía, J., Serrano, E., Pavón, J.: Testing and debugging of MAS interactions with INGENIAS. In: Luck, M., Gomez-Sanz, J.J. (eds.) AOSE 2008. LNCS, vol. 5386, pp. 199–212. Springer, Heidelberg (2009)

[15] Gutknecht, O., Ferber, J., Michel, F.: Integrating tools and infrastructures for generic multi-agent systems. In: AGENTS 2001: Proceedings of the Fifth International Conference on Autonomous Agents, pp. 441–448. ACM, New York (2001)
[16] Kaddoum, E., Gleizes, M.-P., Georg, J.-P., Picard, G.: Characterizing and evaluating problem solving self-* systems. In: Dini, P., Gentzsch, W., Geraci, P., Lorenz, P., Singh, K. (eds.) Computation World: Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns (Adaptive 2009), pp. 137–147. IEEE Computer Society, Los Alamitos (2009)
[17] Kephart, J.O., Chess, D.M.: The vision of autonomic computing. IEEE Computer 36(1), 41–50 (2003)
[18] Kidney, J., Denzinger, J.: Testing the limits of emergent behavior in mas using learning of cooperative behavior. In: Proceeding of the 2006 Conference on ECAI 2006, pp. 260–264. IOS Press, Amsterdam (2006)
[19] Lam, D.N., Barber, K.S.: Debugging agent behavior in an implemented agent system. In: Bordini, R.H., Dastani, M.M., Dix, J., El Fallah Seghrouchni, A. (eds.) PROMAS 2004. LNCS (LNAI), vol. 3346, pp. 104–125. Springer, Heidelberg (2005)
[20] Lomuscio, A., Raimondi, F.: MCMAS: A Model Checker for Multi-agent Systems. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 450–454. Springer, Heidelberg (2006)
[21] Luck, M., McBurney, P., Shehory, O., Willmott, S.: Agent Technology: Computing as Interaction (A Roadmap for Agent Based Computing), AgentLink (2005)
[22] Moreno, M., Pavón, J., Rosete, A.: Testing in agent oriented methodologies. In: Omatu, S., Rocha, M.P., Bravo, J., Fernández, F., Corchado, E., Bustillo, A., Corchado, J.M. (eds.) IWANN 2009. LNCS, vol. 5518, pp. 138–145. Springer, Heidelberg (2009)
[23] Nguyen, C.D.: Testing Techniques for Software Agents, PhD thesis, International Doctorate School in Information and Communication Technologies - University of Trento (2008)
[24] Nguyen, C.D., Miles, S., Perini, A., Tonella, P., Harman, M., Luck, M.: Evolutionary testing of autonomous software agents. In: The Eighth International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2009), pp. 521–528. IFAAMAS (2009)
[25] Nguyen, C.D., Perini, A., Tonella, P.: Ontology-based Test Generation for Multi Agent Systems. In: Proc. of the International Conference on Autonomous Agents and Multiagent Systems (2008)
[26] Nguyen, C.D., Perini, A., Tonella, P.: Goal-oriented testing for MASs. Int. J. Agent-Oriented Software Engineering 4(1), 79–109 (2010)
[27] Núñez, M., Rodríguez, I., Rubio, F.: Specification and testing of autonomous agents in e-commerce systems. Software Testing, Verification and Reliability 15(4), 211–233 (2005)
[28] Padgham, L., Winikoff, M.: Developing Intelligent Agent Systems: A Practical Guide. John Wiley and Sons, Chichester (2004)
[29] Perepletchikov, M., Ryan, C., Frampton, K.: Cohesion metrics for predicting maintainability of service-oriented software. In: QSIC 2007: Proceedings of the Seventh International Conference on Quality Software, pp. 328–335. IEEE Computer Society, Washington (2007)
[30] Perepletchikov, M., Ryan, C., Frampton, K., Tari, Z.: Coupling metrics for predicting maintainability in service-oriented designs. In: Australian Software Engineering Conference, pp. 329–340 (2007)

[31] Poutakidis, D., Winikoff†, M., Padgham, L., Zhang, Z.: Debugging and Testing of Multi-Agent Systems using Design Artefacts. In: Multi-Agent Programming, pp. 215–258 (2009)
[32] Raibulet, C., Masciadri, L.: Towards evaluation mechanisms for runtime adaptivity: from case studies to metrics. In: Dini, P., Gentzsch, W., Geraci, P., Lorenz, P., Singh, K. (eds.) Computation World: Future Computing, Service Computation, Cognitive, Adaptive, Content, Patterns (Adaptive 2009), pp. 146–152. IEEE Computer Society, Los Alamitos (2009)
[33] Serrano, E., Botia, J.A.: Infrastructure for forensic analysis of multi-agent systems. In: Hindriks, K.V., Pokahr, A., Sardina, S. (eds.) ProMAS 2008. LNCS, vol. 5442, pp. 168–183. Springer, Heidelberg (2009)
[34] Serrano, E., Gómez-Sanz, J.J., Botía, J.A., Pavón, J.: Intelligent data analysis applied to debug complex software systems. Neurocomputing 72(13-15), 2785–2795 (2009)
[35] Sierra, C., Aguilar, J.A.R., Noriega, P., Esteva, M., Arcos, J.L.: Engineering Multi-Agent Systems as Electronic Institutions. Novatica 170, 33–39 (2004)
[36] Sudeikat, J., Renz, W.: A systemic approach to the validation of self–organizing dynamics within MAS. In: Luck, M., Gomez-Sanz, J.J. (eds.) AOSE 2008. LNCS, vol. 5386, pp. 31–45. Springer, Heidelberg (2009)
[37] Tiryaki, A.M., Öztuna, S., Dikenelli, O., Erdur, R.C.: SUNIT: A unit testing framework for test driven development of multi-agent systems. In: Padgham, L., Zambonelli, F. (eds.) AOSE VII / AOSE 2006. LNCS, vol. 4405, pp. 156–173. Springer, Heidelberg (2007)
[38] Zhang, Z., Thangarajah, J., Padgham, L.: Automated unit testing for agent systems. In: 2nd International Working Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2007), Barcelona, Spain, pp. 10–18 (2007)
[39] Zhang, Z., Thangarajah, J., Padgham, L.: Automated unit testing intelligent agents in pdt. In: AAMAS (Demos), pp. 1673–1674 (2008)
[40] Zhang, Z., Thangarajah, J., Padgham, L.: Model based testing for agent systems. In: AAMAS, vol. (2), pp. 1333–1334 (2009)

Processes Engineering and AOSE

Massimo Cossentino[1], Marie-Pierre Gleizes[2],
Ambra Molesini[3], and Andrea Omicini[3]

[1] ICAR CNR, Viale delle Scienze, ed. 11, 90128 Palermo, Italy
cossentino@pa.icar.cnr.it
[2] SMAC team, IRIT, University of Toulouse,
118 Route de Narbonne, F-31062 Toulouse Cedex 9, France
Marie-Pierre.Gleizes@irit.fr
[3] Alma Mater Studiorum – Università di Bologna
viale Risorgimento 2, 40136 Bologna, BO, Italy
via Venezia 52, 47521 Cesena, FC, Italy
ambra.molesini@unibo.it, andrea.omicini@unibo.it

Abstract. Agent-oriented methodologies like ADELFE, ASPECS, INGENIAS, MaSE, PASSI, Prometheus, SODA, or Tropos propose development formulae with their own specificities. Analyzing them is the responsibility of the Process Engineering discipline, which is currently one hot research line in software engineering. The analysis makes it possible to construct a catalogue of current processes, assessing their utility and enabling their reuse. Additionally, the study may lead to the modification or improvement of existing development processes, perhaps combining fragments from solutions coming from the different methodologies.

In this paper, we first provide a general view over the area of Software Process Engineering (SPE), then focus on the most recent developments of SPE in the Agent-Oriented Software Engineering (AOSE) field.

Keywords: Agent-Oriented Software Engineering, Methodologies, Software Process Engineering, Fragment.

1 Software Processes

In Software Engineering (SE), developers expects to find concrete "instructions" and methods to complete development before a deadline, delivering a quality product, and with a cost meeting the initial budget of the project. Such instructions are typically expressed in the form of a software *development process*. Development processes have often appeared in Agent Oriented Software Engineering (AOSE) methodologies as a simple enumerated list of steps. While this can be considered effective for small developments – e.g. one person during a short period of time –, it is hardly applicable to medium-to-big developments—e.g. a development team with a project several years long. The literature suggests using more complex solutions, like Cernuzzi et al. [1] or Fuggetta [2].

Cernuzzi et al. [1] define the development process as "an ordered set of steps that involve all the activities, constraints and resources required to produce a

M.-P. Gleizes and J.J. Gomez-Sanz (Eds.): AOSE 2009, LNCS 6038, pp. 191–212, 2011.

specific desired output satisfying a set of input requirements". Fuggetta [2] proposes an interesting definition from the organizational point of view of *software development process* (or simply *software process*) as "the coherent set of policies, organizational structures, technologies, procedures, and artifacts that are needed to conceive, develop, deploy, and maintain (evolve) a software product". Software processes can then be (and typically are) composed by a set of stages, each specifying which phases/activities should be carried on and which roles (i.e.: client, analyst, software architect, programmers, etc.) and resources are to be involved in them.

Such concepts are seldom found in existing agent-oriented methodologies. Typically, new AOSE methodologies get proposed without explicitly relating them to any process models and, at the same time, being implicitly suitable only for a limited set of (or a single) process models. This contrasts with the fact that designing multi-agent systems is a complex software development, and that developers need to be guided and helped in this task. Therefore, agent-oriented methodologies should also take into account the fact that the product is not merely developed but also: (i) conceived, often relying on unstable or incomplete requirements; (ii) deployed, i.e., put to work in an operational environment; (iii) maintained and evolved, depending on novel requirements or changes in the operational environments. Besides, intuitively, different needs call for different processes. For example, it is quite clear that the way a multi-agent system (MAS) dedicated to solve a specific problem such as a timetabling [3] and a MAS which has to simulate a natural phenomenon [4] involve different development approaches.

Another important aspect to be considered is that a software process is not something static that, once adopted, should never be changed. Instead, a process can be improved with the experience of applying it in a concrete development. For instance, PASSI has been modified in order to also handle multi-agent simulation [5], while Tropos and ADELFE are combined in order to take into account self-adaptation in Tropos [6]. As a consequence, any agent-oriented software engineer should assume the development process evolves over time. It evolves together with the increased awareness of the involved people, towards a maturity level[1] that ensures the repeatability of a process in terms of the quality, cost and time of the produced software. This is a fundamental evaluation criterion for any organization that would aim at adopting the agent-based paradigm in its development process.

As a conclusion, the development process is an important element of any software engineering methodology—and is essential within any agent-oriented methodology; it attends the needs of a concrete development problem and evolves when the change brings benefits. Therefore, the study of the many aspects of the software development process from a scientific perspective is mandatory.

Along this line, in this paper we adopt *Software Process Engineering* (SPE) as the conceptual and technical framework for such a study, then discuss its application to AOSE. In short, SPE provides the tools for analyzing, decomposing, and

[1] CMMI, http://www.sei.cmu.edu/cmmi/tools/cmmiv1-3/

building software development processes. Accordingly, Section 2 briefly describes SPE under the classical SE perspective, along with the main concepts used in process composition. Then, Section 3 presents SPE from the AOSE point of view, and expounds the recent works done on fragments and process engineering customization.

2 Software Processes Engineering

Like most processes concerning human activities, the software development process is inherently complex and hard to standardize, as demonstrated by the difficulties reported in the literature of automating the software process. Despite the great acceptance of development processes based on the Unified Process (UP [7]), researchers and practitioners in the area agree that there is no single software process that could fit the need of all [8]. UP itself is essentially a highly-customizable development framework rather than a well-established and fixed process.

Among the current method engineering approaches in the literature, Situational Method Engineering (SME) is undoubtedly one of the most promising: there, the construction of ad-hoc processes is based on reusing portions of existing or newly-created processes, called *fragments*. So, in the reminder of this section we first discuss the basis of SME (Subsection 2.1), then we sketch the diverse fragment definitions (Subsection 2.2) and fragment repositories (Subsection 2.3), finally we briefly introduce some tools supporting SME (Subsection 2.4).

2.1 Situational Method Engineering

Situational Method Engineering [9,10] is the discipline that studies the composition of new, ad-hoc software engineering processes for each specific need. This is based on the assumption that the "situation" is the information leading to the identification of the right design approach. This paradigm provides tools for the construction of ad-hoc processes by means of the reuse of existing process fragments (called method fragments), stored in a repository (Subsection 2.3). In order to be effective, this approach requires the process and its fragments to be suitably modeled.

Within the concept of situation authors usually include: requirements of the process, development context, the specific class of problem to be solved. Several approaches to SME have been presented in the last years with no specific reference to the OO context [8,11,12,13,14].

The most complete approach in the field is probably represented by the OPEN Process Framework (OPF [15]). OPF is based on a large number of method fragments stored in a repository along with a set of construction guidelines that are considered to be parts of existing methodologies and can be used to construct new methodologies. The OPF meta-model is composed of five main meta-classes [16]: Stages, Producers, Work Units, Work Products and Languages. When instantiated, each meta-class produces a method fragment.

2.2 Fragment Definition

Different approaches to process composition may be also rely on different definitions (and labels) for the building blocks. We may attempt a rough categorization of process reusable parts defined in classical SE:

Method Fragment — A method fragment is a piece of an information systems development process. According to Brinkkemper et al. [11,17], there are two kinds of method fragments: the product fragment – concerning the structure of a process product, representing deliverables, diagrams, ... – and the process fragment—describing the stages, activities and tasks to be performed to produce a product fragment.

Method Chunk — According to J. Ralyte et al. [12,18], a method chunk is a consistent and autonomous component of a development process. The method chunk integrates two aspects of the method fragment, the product and the process, so it represents a portion of process together with its related product(s).

OPF Method Fragment — An OPF method fragment is an instance of one of the following classes: Stages, Producers, Work Units, Work Products and Languages. According to D. Firesmith and B. Henderson-Sellers [15,19], fragments of different types need to be composed in order to provide the different features of a process.

2.3 Fragment Repository

A *fragment repository* (or *method library*) is an essential component of any SME approach. In spite of this, only few repositories are currently available, due to the great effort required to build such a resource as well as to the lack of a widely-accepted standard in the field.

A repository could be used as a general purpose software engineering processes container. It could even contain sets of reusable elements (content and process) that do not belong to a specific software engineering processes, but could be used as building bricks (fragments/chunks). The largest available repository is part of OPF [15][2]. The support for a method library is also present in a widespread software tool, the Eclipse Process Framework[3] (EPF) plugin, which supports the OMG Software Process Engineering Metamodel (SPEM) v. 2.0 [20], and provides all the capabilities needed to define method plugins into library as well as to tailor new software engineering processes from those plugins (method configuration). A project such as OpenUP[4] provides an open-source, common and extensible base for iterative incremental software engineering processes by using the modularity and re-usability skills of SPEM 2.0 through the EPF plugin.

[2] http://www.opfro.org/
[3] http://epf.eclipse.org/
[4] http://epf.eclipse.org/wikis/openup

2.4 Tools

When developing a software system, several tools are required to support the different stages of the process. Areas of application for design tools spread from requirements elicitation to design, testing, validation, version control, configuration management and reverse engineering. In this section, not all tools available during the engineering process are described. We rather would like to focus on the different kinds of tools linked to apply or adapt a method. Tools can be classified in three different categories: CASE, CAME, and CAPE tools.

The CASE acronym stands for Computer Aided Software Engineering, and it addresses the large category of tools that could be used in any software engineering process employment. CASE tools support the modeling activities and constrain the designer just in the choice of the system modeling language (for instance UML), and, when code generation is possible, on the coding languages. The main limit of these tools is that they are not aware of the adopted method – in terms of work to be done – and are instead only concerned with the representation of some (often uncoordinated) views of the system.

Today, in the field of aided software development, meta-CASE tools achieved large significance, being able to provide an either automated or semi-automated process for the creation of CASE tools, based on a meta-model used to describe the language, the concepts and the relationships of a specific design methodology. In the field of Method Engineering a meta-CASE tool – called CAME (Computer Aided Method Engineering) – is used to describe and to represent a set of methods. CAME tools are conceived to support methods rather then design. They do not adopt any specific software development process model – they are not even aware of its existence because they are only concerned with the drawing of the different aspects of the model separately –, therefore the designer could freely work on the different views, even violating the prescribed process without any warning from the tool.

Several existing CAME tools (Mentor [21], Decamerone [22], MetaEdit+ [23], INGENME[5] and MethodBase [24]) are also based on Meta-CASE technology allowing the construction or the automatic generation of CASE tools specific for given methods. In particular, MetaEdit+ seems the most complete CAME tool, even if it presents several limitations. MetaEdit+ makes it possible to implement a domain specific modeling language, and to use it in ad-hoc created CASE tool. MetaEdit+ is at the same time a CAME and a CASE tool, and is the only tool that allows the instantiation of a CASE tool starting from the definition of a given modeling language. However, MetaEdit+ does not provide any support for managing the development processes, and the generated CASE tools always adopt the same UML editor for the software product design. As a result, the generated methods are static, and there is no way to reuse portions of existing tools supporting the method fragments used. A possible alternative to MetaEdit+ is INGENME, a tool for producing self-contained visual editors for languages defined using an XML file. INGENME can be used to test visual languages and also to produce customized editors.

[5] http://ingenme.sourceforge.net/

One of the intrinsic limits of CAME tools – the lack of process awareness – is overcome by CAPE (Computer Aided Process Engineering) tools that are instead aware of the adopted process model – or, that could be used to design it –, and coordinate the different stages of the process in order to respect its prescriptions.

3 AOSE Software Processes Engineering

Putting the focus of process engineering on the development of multi-agent systems moves the attention of the designer to the study of specific topics such as:

- the definition of the agent concept (that is often specific to each single approach),
- the definition of the MAS meta-model (composed of the already cited agent as well as many other entities like role, communication, group, and so on),
- the agent's reasoning technique (goal-oriented, expert system-based, rule based, ...),
- the implementation of autonomy, self-organization and similar system features.

Such topics affect the conception of the AOSE design process. Just to provide an example, it is nowadays very uncommon to find the adoption of formal languages in the development of object-oriented system. Conversely, this is quite frequent in conception of MAS where such languages provide a good support for the design of some reasoning techniques. This obviously influences the developing process since there is the need of several different new activities such as model checking and verification.

Besides, the absence of a standardized and widely accepted MAS meta-model has deeply influenced the introduction of SME techniques in the AOSE field. As a consequence, several authors centered their approach on the influence that the definition of a specific MAS type can have on the process that have to be followed in order to analyze and design such a MAS type. More details about such a variant of the classical SME approach are provided in the next subsection.

3.1 AOSE Situational Method Engineering

As far as the approaches specifically related to the agent-oriented context are concerned, an initial reference framework has been provided by the work of the FIPA Methodology Technical Committee (TC)[6] devoted to the study of fragments' definition and composition [25]. A standard specification is expected by the successor of that committee, the IEEE FIPA Design Process Documentation and Fragmentation (DPDF) working group[7]. The DPDF WG approach is based

[6] http://www.fipa.org/activities/methodology.html
[7] http://www.fipa.org/subgroups/DPDF-WG.html

on the adoption of SPEM 2.0 with the inclusion of minor extensions motivated by the specific needs of a multi-agent system design process [26].

SME in AOSE shares the same objective with SME researchers in proposing the most relevant process for a given situational context of development. The objective is to provide CAPE tools enabling to build the most convenient process and possibly to adapt it during the development. In fact, the most relevant fragments must be retrieved and used among all the available fragments. Consequently, some of the works in the AOSE community currently focus on how to automate the software process construction such as the three following examples reported here.

PRoDe: A Process for the Design of Design Processes. PRoDe [27] is an approach for new AOSE design process composition based on two pillars: the process fragment [28] and the MAS meta-model. These two elements are both defined and considered under a specific agent-oriented perspective thus creating a peculiar approach. PRoDe is based on the classic situational method engineering and it is organized in three main phases: Process Analysis, Process Design and Process Deployment. They clearly resemble the software development phases thus realizing the parallelism proposed by Osterweil in his well-known paper *Software Processes are Software too* [29].

During Process Analysis, the process to be developed is analyzed and its requirements are elicited. Process Requirements Analysis delivers a portion of the MAS meta-model, whose structure will decisively affect the following steps of process development. During Process Design the method engineer selects, from a previously constructed repository, a set of fragments that he/she assembles in the new process. Finally, in the Process Deployment phase, the new process is used to solve a specific problem. From this employment some feedbacks are received and this is used to further enhance the process towards its maturity. It is also possible to repeat the whole construction process in an incremental/iterative way.

The most relevant contribution coming from PRoDe to the state of the art consists in some guidelines driving the method designer through the most difficult steps of the work. This is a very relevant aspect of the approach since skills required to a method engineering are very high and such a professional profile is not frequently found in the field. The most relevant guideline is realized by an algorithm used to prioritize the retrieval of fragments from the repository. The problem solved by the algorithm is: given a MAS meta-model and the set of fragments able to instantiate the elements of the meta-model, which is the first fragment that should be selected? And the following? This problem is relevant because the selection of one fragment (the first) introduces new constraints in the design: its inputs are to be satisfied and its outputs should be all used otherwise the fragment needs an adaptation (an adjunctive cost).

Currently the process has been already adopted in several case studies [30,31] and it is at the basis of the approach proposed in the next subsection.

MEnSA Project. In the MEnSA project[8] the authors decided to directly link the new process requirements to the fragments they were going to select. The composition of the new methodology was inspired by the PRoDe approach [27], which proposes to use the MAS meta-model as a central element for selecting and assembling fragments. It is worth to note that while in the PRoDe approach requirements are used to compose an initial draft of the MAS meta-model, which is then used to retrieve fragments from the repository, in the novel MEnSA approach process requirements are used to select fragments, and their outcomes are used to define the MAS meta-model.

The approach adopted is organised in a few steps. The first step of the work consists in collecting process requirements, currently some methodologies have been decomposed in fragments: PASSI, Gaia, Tropos and SODA. Then, fragments are retrieved from the repository according to the requirements they contribute to fulfill.

The *fragments selection* activity makes available the set of fragments used to produce a first draft of the MAS meta-model. Thus, each fragment contributes to define a portion of the meta-model.

Once the meta-model has been polished, the initial set of fragments finds its position in a proper life-cycle, therefore a proper process model has to be chosen. Classically-available life-cycles (waterfall, iterative/incremental, spiral, etc.) are here considered, and the best fitting is used to host the selected fragments.

Now fragments can be positioned in the life-cycle placeholders, and a first version of the new process is almost ready. The last activity is *fragments adaptation*, which aims at solving incompatibility issues arising fragments from different processes. Such fragments should then be adapted to properly support the new MAS meta-model and to comply with all input/output constraints.

At this stage, an initial version of the process is finally available. This could be either complete or incomplete according to the number and refinement of the initial process requirements, as well as to other factors—fragment repository dimension, assembly issues, process phases coverage, ...

In the MEnSA process composition, when the process needs to be completed, the authors follow up with an iteration of the proposed composition approach, given its smaller granularity and its MAS-meta-model-based approach that perfectly fits the needs of the new process final refinement.

Self-Organisation-based SPE Design. The aim of this approach is to provide a system able to combine existing fragments to build a software process adequate to the expertise level of designers and to the applications features as well. The designer may interact with the system in order to modify the software process proposed by the system. The SPE is co-constructed by the system and the designer.

The system continuously self-adapts to these new perturbations and proposes a new software process taking into account the designer's wishes. Each fragment

[8] Methodologies for the Engineering of complex Software systems: Agent-based approach. For more details, see the project website at: http://www.mensa-project.org/

is *agentified* following the Adaptive Multi-Agent Systems (AMAS) theory [32]. The adaptive MAS automatically designs an adaptive software engineering process [33]. The resulting MAS is composed of (i) the fragments, which are the sole agents of the MAS (called fragment-agents), and (ii) the resources of the MAS, which are the MMMEs (MAS Meta-Model Elements), the MMME repository and the fragments repository.

The first MAS prototype is developed using a repository containing the already-defined fragments of three processes from three methodologies: ADELFE [34], INGENIAS[9], and PASSI [35].

In this system, fragments are autonomous agents able to find other relevant fragments to interact (workproduct exchanges). The fragment-agents cooperate with each other in order to find their right location in the software process. The agentification of fragments is realized in a general way to enable the adding or removal of a fragment in the set of all fragments. In the first version of the system, agentification must ensure a mutual understanding between what is required by a fragment and what is produced by it—i.e., the agent-fragments understand each other.

The behavior of a fragment is represented by a finite state automaton with three states: *Inactive*, *Unsatisfied* and *Satisfied*. The fragment-agent switches states according to its perception of the environment and behaves according to its current state. Following the AMAS theory, the design of agents focuses in particular on the Non Cooperative Situations (NCS) [32]. Encountering a NCS is one of the possible reasons for a fragment-agent to change its state. For a fragment-agent, three NCS are identified. The first is when none of the MMMEs that a fragment-agent can produce are needed by the other fragments (the agent is *useless*). The second is when a fragment-agent needs some MMMEs that no other fragment-agent can produce (the agent *cannot satisfy its preconditions*). The third is when two fragment-agents of the same process produce the same MMMEs (agents are in conflict, and must determine which one of them is *useless*). To react to such NCS, a fragment-agent can execute three main actions. First, it can use existing MMMEs to produce new ones and register them to the repository. Or, it can stimulate other fragments able to produce their needed MMMEs by sending a stimulation value. This value depends on the amount of stimulation it received. So, the stimulation value of a fragment-agent can be assimilated to a measure of its criticality, since the more it is important to the system, the more a fragment-agent will be stimulated. Finally, it can observe other agents and determine its membership to a process.

3.2 Fragment Definition

As far as the agent-oriented approaches are concerned, some contributions for the adoption of the OPF framework [36] to agent design have been proposed such as in PASSI [37] and Tropos [38]. A different solution was proposed by the FIPA Methodology TC for the adoption of a kind of method chunk, called *process fragment*, which is a portion of a development process defining deliverables

[9] http://grasia.fdi.ucm.es/main/node/241

(workproducts), guidelines, preconditions, sets of system meta-model elements and key-words characterizing it in the scope of the method in which it was defined. The main difference with respect to the OPF lays in the focus of the MMMEs managed by the process fragment. The definition and use of a MAS meta-model enables the adoption of Model-Driven Engineering practices, and overcomes typical problems due to the confusing definition of agent-oriented concepts in most AOSE methodologies. Besides, considering fragments through such a rough definition eases the adoption of SPEM 2.0 concepts as well as a high level of compliance to that standard. However, [25] presents an enhanced version of the fragment meta-model originated by the FIPA Methodology TC; one of the proposals is that fragments should be considered from different points of view whether the designer is interested in their reuse, storing, implementation, or in the definition of the process they represent. So, it seems obvious that the choice of a process fragment depends on the point of view and the requirements it has to fulfil.

Process fragments do not match a single SPEM 2.0 concept, instead they should rather be considered as matching several ones depending on the granularity or concern. A process fragment is a portion of a process, but process element definitions in SPEM 2.0 are divided into two categories: definition and use. Thus, such a separation should be taken into account while mapping fragments to SPEM 2.0 concepts. Furthermore, also fragments granularity should to be considered, as it implies a different mapping.

It is worth to note that although the use of SPEM is widespread in the SE community, a new standard has been recently published by ISO [39,40], which models both the design and enactment of a process by using a multilayered architecture. The peculiarity of that work lays in the adoption of *powertypes*, an innovative mechanism allowing a class to assume value for its attributes not necessarily when instantiated in the next abstraction level but, if required, at the second level [41]. A similar standardization effort, in the AOSE field, is currently ongoing within the IEEE FIPA DPDF working group. The first step has been the identification of the most suitable process meta-model and notation: (i) for the representation of the existing design processes from which the fragments have to be extracted, and (ii) for the representation of fragments themselves. An important contribution on the subject might come from the SPEM 2.0 specification.

The second step has been consisted in the definition of a proper template for the description of agent-oriented design processes. Such a template refers to the selected process meta-model and suggest the adoption of good practices in documenting existing processes as well as defining new ones. The availability of such a specification would have several benefits, the first is that adopting the same documentation template would enable an easier comprehension (and comparison) of existing and new processes. This goes in the direction initially drawn by UML creators (Booch, Jacobson, and Rumbaugh) when they first removed all the specific notation issues cluttering their own approaches and then easily found commonalities that inspired their new (and successful) modeling

language [42]. The specification for process documentation has been already proposed for adoption as an IEEE FIPA standard.

After that, the group is going to define the process fragment structure and according to that a procedure for extracting fragments from processes documented according to the adopted template. The final result will consist in a set of fragments that are compliant to the fragment specification and are documented according to the same style. This would enable their composition and the production of a consistent documentation of the new process.

3.3 Fragment Repository and Tools

Some extensions of the OPEN framework [15] aiming at including the support for agent-oriented methodologies have been recently presented in [36,37,38,43,44,45]. Another (explicitly agent-oriented) repository[10] has been developed according to the specifications proposed by the FIPA Methodology TC [46], used as a starting point by the IEEE FIPA DPDF working group.

The proposed repository structure is an XML-based repository storing a collection of XML documents, each one representing a method fragment, validated by a Document Type Definition (DTD) or an XML Schema [25]. The validation process ensures that the method fragment was extracted and defined according to the prescribed meta-model (Subsection 3.2). The repository is oriented towards a MAS meta-model-based classification of fragments; each one of them is in fact labeled with the MAS meta-model components that are defined or refined during its activities. Each activity has some inputs and produces some outputs in terms of defined/refined components of the MAS meta-model [25]. At the moment, the repository contains the fragments coming from PASSI, ADELFE and Tropos. Also, AOSE could benefit from projects such as OpenUP by defining specific AO method plugins and reusing predefined ones.

As far as tools are concerned, some of the CAME tools introduced in Subsection 2.4 – such as INGENME – are general enough to be effectively reused in the AOSE context. In addition, an example of CAPE tool specifically developed in the agent field is represented by Metameth [47], which allows the composition of a set of fragments stored in a specific repository. Metameth seems the only tool considering interactions with existing external tools for the creation of a CASE tool based on the characterization of the interaction between Metameth and the external tools. At the moment, these kinds of interactions are rather limited because of the complexity of the composition of the existing tools' portions—which are typically based on different development principles and specific APIs, to be taken into account in order to compose at the best the different tools.

3.4 AOSE Meta-model

In their most general acceptation, meta-models have been addressed from different points of view. Just to cite some of them we can list what follows:

[10] http://www.pa.icar.cnr.it/passi/FragmentRepository/fragmentsIndex.html

Bernon et al. — The process of designing a system (object or agent-oriented) consists of instantiating the system meta-model that the designers have in their mind in order to fulfill the specific problem requirements. In the agent world this means that the meta-model is the critical element because of the variety of methodology meta-models [48].

Gonzalez-Perez et al. — A meta-model is a model of a methodology or, indeed, of a family of related methodologies [49].

Brian Henderson-Sellers — A meta-model describes the rules and constraints of meta-types and meta-relationships. Concrete meta-types are instantiated for use in regular modeling work. A meta-model is at a higher level of abstraction than a model. It is often called *a model of a model*. It provides the rules/ grammar for the modeling language itself. The modlling language consists of instances of concepts in the meta-model [50].

Although it is possible to describe a methodology without an explicit meta-model, formalizing the underpinning ideas of the methodology in question is valuable when checking its consistency or when planning extensions or modifications. The importance of meta-model becomes clear when it is necessary to study the completeness and the expressiveness of a methodology, and when comparing different methodologies. Consequently, there is the need to study the different AOSE methodologies, to compare their abstractions, rules, relationships, and the process they follow, all of that would lead to a more comprehensive view of this variety of methodologies. Different works have been devoted to the study [51,52,53,54,55,56,57], and the unification of MAS meta-models [48,50,58,59], that it has been one of the more important topic in the work done by the agent community within the Agentlink Agent-Oriented Software Engineering Technical Forum Group (AOSE TFG) meetings[11].

The importance of meta-modeling is not only for having a precise view of agent-oriented methodologies as a way to check their completeness and expressivity, or to compare them, but it is also useful to clarify the distance between agent-oriented methodologies and infrastructures [51]. Expressing agent-oriented methodologies and infrastructures through formal meta-models is an initial step to reduce the conceptual and technical gap amongst these two research areas[12].

The situation up-to-date is that the lack of a unique MAS meta-model leads each methodology to deal with its own concepts and system structure, even if currently there are two kind of standardization efforts. The first effort is the recent standard "Software Engineering Metamodel for Development Methodologies" ISO/IEC 24744 [13] [60]. This standard is not specific for the agent-oriented field, rather it is very general. It is based on two powerful but someway complex concepts: powertype, which was already introduced, and clabject, i.e. a class/object hybrid concept. These tow concepts could easily lead to reduce the understandability of the standard.

[11] See http://www.pa.icar.cnr.it/cossentino/al3tf3/ for more details.
[12] http://www.mensa-project.org/
[13] See http://www.iso.org/iso/catalogue_detail.htm?csnumber=38854

The second effort is represented by the OMG Agent Platform Special Interest Group[14] (Agent PSIG) that tries to identify and recommend new OMG specifications in the area of agent technology, with particular attention to:

- recommend agent-related extensions to existing and emerging OMG specifications;
- promote standard agent modeling languages and techniques that increase rigor and consistency of specifications;
- leverage and interoperate with other OMG specifications in the agent area.

The work of the Agent PSIG is still in its infancy and at the time of writing there are no specific results nor public documents.

3.5 Agent-Oriented Design Processes

In this section, we will provide a quick survey on some processes from literature. In order to provide a schematic view we will compare them according to a list of features. The list of compared AOSE processes includes: ADELFE [61], ASPECS [30], GAIA [62], INGENIAS [63], MaSE [64], PASSI [65], Prometheus [66], SODA [67,68].

A lot of approaches have been published about processes comparison (for instance see: [69,70,71,72,73,74]) and we decided to adopt a plain one based on a the consideration of a minimal set of process features. For each feature, we examine if the process exhibits it or not; the list of features is described below:

Coverage of the Entire Lifecycle — The methodology designer should be interested in developing a detailed and complete methodology from existing analysis till software deployment and maintenance. It is now widely accepted that the core workflows of a methodology are requirements collection, analysis, design, development (also called implementation), deployment and testing. Some methodologies cover the entire process of software engineering while some others are more focused on a part of that.

Problem Type — The challenge of agent or multi-agent design processes is to help the designers in building complex systems such as multi-agent systems usually are. Two categories of works could be found: those which are general high-level methodologies, and those which a focus on a specific application context.

Underlying Agent Architecture — Different design processes often refer to different agent architecture. Some processes look at BDI agents, some others refer to the IEEE FIPA specifications and finally there are processes that are not necessarily related to a specific architecture.

Origin — Some agent-oriented design processes are based on concepts and theories developed in the object-oriented field. Sometimes they even explicitly refer to an existing OO process (such as the Unified Process, UP).

[14] http://agent.omg.org/

Notation — Several design processes propose a proprietary notation to be adopted in the work products they specify. In some cases notation is an application or an extension of well known modeling languages (mainly UML), in other situations notation is a specific one, in one case there even is the proposal of a shared adoption of the same notation (as it happens in [75]).

Lifecycle — While modern object-oriented processes like UP are nowadays flattered to the adoption of the incremental/iterative lifecycle, agent-oriented approaches are still variegate and exhibit different solutions. Actually old paradigms like waterfall are still adopted as well as the last iterative/ incremental and even agile ones (not adopted by any of the here discussed processes but by some others).

Support for Model Transformations — The acceptance of a model-driven engineering paradigm [76] is growing in popularity in the agent-oriented landscape. According to that, (portions of) design models are obtained by transformation of preceding ones thus reducing design effort and at the same time increasing design quality. Several agent-oriented design process have been conceived to support that at a different extent.

Design Support Tools — The availability of a specific support tool allows for an easier enactment of the design process and usually increases the resulting design quality thanks to a set of check on notation and semantic aspects of the models.

Results of the comparison conducted on the basis of these criteria are reported in Table 1. Other design processes are reported in literature although they have not been compared with the previous ones. Among the others we may list: Agent-PriME [77], ASEME [78], AOR [79], Gormas [80], MESSAGE [81], O-MaSE [57], Tropos [82].

3.6 Roadmap

Multi-agent systems are increasingly used in different kinds of applications. Designing such systems requires handling different points of view about the system to be done. Sometimes, the correct way to handle these perspective have just to be discovered. For instance, ant-based simulation requires to take into account different environments: the simulation environment composed in the simulation participants; the client who provides the data and has to analyze and observe the running system; and the MAS environment which is composed of resources accessible by agents. Instead of building new development methods to deal with these three environments, it seems more logical to investigate how one existing method can be modified to take them into account, such as: PASSI which is slightly modified to take into account simulation requirements[5].

Fragments represent a paradigm which enables to diminish the cost of method definition by reusing existing ones. Although a couple of repositories already exist, much work is still to be done in the field. Little experience exists on widespread design processes composition and there is an obvious relationship between the quality of fragment repository and the assembled process. Moreover, the opposite is true as well. The more processes will be composed by using

Table 1. A comparison of some agent-oriented design processes. $^{(a)}$ Unified Notation proposed in [75].

Criterion	ADELFE	ASPECS	GAIA	INGENIAS	MASE	PASSI	PROMETHEUS	SODA
Entire Life-cycle	Yes	Yes	No	Yes	No	Yes	No	No
Problem Type	Open systems, Dynamic environment	Hierarchical decomposable problems	Open systems	Not specified	Robotic Systems	Information Systems, Robotic Systems	General Purpose	Open systems
Agent Architecture	Cooperative Agents	Holonic agents	Not specified	BDI	Not specified	FIPA	BDI	Not specified
Origin	UP	PASSI, RIO	Not specified	UP	UP	UP	Not specified	Not specified
Notation	UML Extension	UML adaptation	Non specific	Specific notation	UML adaptation$^{(a)}$	UML adaptation$^{(a)}$	UML adaptation$^{(a)}$	Tabular
Life-cycle Model	Iterative	Iterative / Incremental	Waterfall	Iterative	Iterative	Iterative / Incremental	Iterative	Iterative
MDE Support	Yes	Yes	No	Yes	No	Yes	No	Yes
Tools	Yes	No	No	Yes	Yes	Yes	Yes	No

fragment reuse, the more experience will be available on the field thus enabling an improvement of fragment definitions and repository structures. The main future research axis has to focus on three main elements: a language for fragments description; the means to ensure the interoperability between fragments; and tools to facilitate the fragments composition in order to produce the relevant software processes. A language is needed that guarantees interoperability of fragments developed by different designers and for different purpose. A main obstacle towards this interoperability is the MAS meta-model. Fragments of processes are associated with fragments of MAS specifications. Hence, a fragment depends on a MAS meta-model. The lack of a unified MAS meta-model topic has been long discussed in the AOSE community (just think about the debates held during the Agentlink AOSE TFG events[15]) and nonetheless it is still an unsolved issue. Besides the underlying MAS meta-model, the fragments must work with each others and the problem here is quite closed to component architecture. In any case, the work on languages for fragments description requires necessarily the assistance of CAPE tools which validate the devised solutions. CAPE tools permit designers to build the most relevant software processes regarding the application and the designers team expertise. Several steps can be followed: from the hand-made static built process to the self-design process; and from process determined at the beginning of the software development to a dynamic process adapted to the development status.

Other open research issues will briefly discussed below:

- The definition of application specific processes. Even considering the amount of existing processes as a good starting point for several custom products, there still is the need for specific approaches related to specific domains.
- The integration of agents with services and the related (web service) technology. Web services are nowadays widely spread and represent a good and affordable solution for the development of highly distributed systems. Where can agents contribute in a development scenario dominated by services? Probably the answer lays in the essence of agency: agents are autonomous, proactive, and social. These properties are not necessarily shared by services and they provide an invaluable support in the creation of a new abstraction and design layer.
- Agent reasoning techniques. They are not a new issue but they are always an important topic. Too many times agents are realized as simple statecharts. Introducing advanced reasoning capabilities in agents is a complex task but this is at the same time one of the crucial factors for distinguishing agent-oriented systems from traditional ones.
- Common pitfalls in design processes. Testing, deployment and formalized design techniques (patterns) issues have not received, in the agent-oriented field, the same attention they received in classic software engineering despite they are worth to. Testing techniques are often taken from object-oriented systems and adapted with minor changes. What about testing the successful implementation of a certain degree of autonomy or self-organization? No

[15] http://www.pa.icar.cnr.it/cossentino/al3tf3/

definitive answer still exist. Deployment is totally neglected by most of agent-oriented approaches but this should be one of the strength points of MAS (because of their easy distribution). In the era of cloud-computing, the agent community has a great opportunity. Agents may take profit of the elaboration infrastructure proposed by this paradigm and conversely may offer to cloud-computing new ideas for load balancing and distribution. Finally very few agent designers accept the use of design patterns as a daily practice. Several papers have been written on the matter and some pattern repositories exist in literature but the diffusion of them in process employment (but also conception) is still limited.

– Agent modeling languages. This is another long lasting issue. It is strictly related to the research about MAS meta-model but the perspective may be different. Defining a MAS meta-model means defining what are the elements that will be instantiated in the design of a new MAS. Defining a MAS modeling language means defining how the instances will be represented at design time. The link between the two is tight but there is not (necessarily) a one to one link.

4 Conclusion

SPE can bring a number of benefits to AOSE. One is the capability of critically analyzing our own development process by decomposing them into fragments. Another one is enabling the construction of new or altered development process as our knowledge of the needs of concrete domain problems and the performance of applied development processes grow.

While the research on SPE is lively in the general area of software engineering, the complexity of the processes to be modeled and built make the agent-oriented framework possibly the most suitable place for new approaches and solutions. In the AOSE field, current research in SPE aims at providing more flexible software processes taking into account the work already done by AOSE methodology designers, by re-using the most relevant parts (fragments) of any methodology.

At mid-term, research on SPE will likely be concerned with the development of tools supporting SPE, whereas the long-term perspective looks towards a self-designed software processes. Along these lines several difficulties should be overcome, among which the interoperability of fragments and the situational context representation and use seem to be the main ones.

References

1. Cernuzzi, L., Cossentino, M., Zambonelli, F.: Process models for agent-based development. Journal of Engineering Applications of Artificial Intelligence 18(2), 205–222 (2005)
2. Fuggetta, A.: Software process: a roadmap. In: ICSE 2000: Proceedings of the Conference on The Future of Software Engineering, pp. 25–34. ACM Press, New York (2000)

3. Picard, G., Bernon, C., Gleizes, M.P.: Etto: Emergent timetabling organization. In: [83]
4. George, J.P., Peyruqueou, S., Regis, C., Glize, P.: Experiencing self-adaptive mas for real-time decision support systems. In: Int. Conf. on Practical Applications of Agents and Multiagent Systems (PAAMS 2009). Springer, Heidelberg (2009)
5. Cossentino, M., Fortino, G., Garro, A., Mascillaro, S., Russo, W.: Passim: A simulation-based process for the development of multi-agent systems. International Journal on Agent Oriented Software Engineering, IJAOSE (2008)
6. Morandini, M., Migeon, F., Penserini, L., Maurel, C., Perini, A., Gleizes, M.P.: A goal-oriented approach for modelling self-organising MAS. In: Aldewereld, H., Dignum, V., Picard, G. (eds.) ESAW 2009. LNCS, vol. 5881, pp. 33–48. Springer, Heidelberg (2009)
7. Jacobson, I., Booch, G., Rumbaugh, J.: The unified software development process. Addison-Wesley Longman Publishing Co., Inc., Boston (1999)
8. Cockburn, A.: Selecting a project's methodology. IEEE Software 17(4), 64–71 (2000)
9. Kumar, K., Welke, R.: Methodology engineering: a proposal for situation-specific methodology construction. In: Challenges and Strategies for Research in Systems Development, pp. 257–269 (1992)
10. ter Hofstede, H.A.M., Verhoef, T.F.: On the feasibility of situational method engineering. Information Systems 22(6/7), 401–422 (1997)
11. Brinkkemper, S.: Method engineering: engineering the information systems development methods and tools. Information and Software Technology 37(11) (1996)
12. Ralyté, J., Rolland, C.: An assembly process model for method engineering. In: Dittrich, K.R., Geppert, A., Norrie, M.C. (eds.) CAiSE 2001. LNCS, vol. 2068, pp. 267–283. Springer, Heidelberg (2001)
13. Henderson-Sellers, B.: Process Metamodelling and Process Construction: Examples Using the OPEN Process Framework (OPF). Annals of Software Engineering 14(1), 341–362 (2002)
14. Hamsen, A.: Situational Method Engineering. Moret Ernst & Young (1997)
15. Firesmith, D., Henderson-Sellers, B.: The OPEN Process Framework: An Introduction. Addison-Wesley, Reading (2002)
16. Gonzalez-Perez, C., McBride, T., Henderson-Sellers, B.: A metamodel for assessable software development methodologies. Software Quality Journal 13(2), 195–214 (2005)
17. Brinkkemper, S., Saeki, M., Harmsen, F.: Meta-modelling based assembly techniques for situational method engineering. Information Systems 24 (1999)
18. Ralyté, J.: Towards situational methods for information systems development: engineering reusable method chunks. In: 13th International Conference on Information Systems Development. Advances in Theory, Practice and Education, pp. 271–282 (2004)
19. Henderson-Sellers, B., Serour, M., McBride, T., Gonzalez-Perez, C., Dagher, L.: Process construction and customization. Journal of Universal Computer Science 10(3) (2004)
20. OMG: Software process engineering metamodel. Version 2.0. Object Management Group (2007)
21. Si-Said, S., Roland, C., Grosz, G.: Mentor: A computer aided requirements engineering environment. In: Constantopoulos, P., Vassiliou, Y., Mylopoulos, J. (eds.) CAiSE 1996. LNCS, vol. 1080, pp. 22–43. Springer, Heidelberg (1996)
22. Harmsen, A., Ernst, M., Twente, U.: Situational Method Engineering. Moret Ernst & Young Management Consultants (1997)

23. Brinkkemper, S., Saeki, M., Harmsen, F.: A method engineering language for the description of systems development methods. In: Dittrich, K.R., Geppert, A., Norrie, M.C. (eds.) CAiSE 2001. LNCS, vol. 2068, pp. 473–476. Springer, Heidelberg (2001)
24. Saeki, M., Iguchi, K., Wen-yin, K., Shinohara, M.: A meta-model for representing software specification & design methods. In: Proceedings of the IFIP WG8.1 Working Conference on Information System Development Process, pp. 149–166. North-Holland Publishing Co., Amsterdam (1993)
25. Cossentino, M., Gaglio, S., Garro, A., Seidita, V.: Method fragments for agent design methodologies: from standardisation to research. International Journal of Agent-Oriented Software Engineering (IJAOSE) 1(1), 91–121 (2007)
26. Seidita, V., Cossentino, M., Gaglio, S.: Using and extending the SPEM specifications to represent agent oriented methodologies. In: Luck, M., Gomez-Sanz, J.J. (eds.) AOSE 2008. LNCS, vol. 5386, pp. 46–59. Springer, Heidelberg (2009)
27. Seidita, V., Cossentino, M., Galland, S., Gaud, N., Hilaire, V., Koukam, A., Gaglio, S.: The metamodel: A starting point for design processes construction. International Journal of Software Engineering and Knowledge Engineering (IJSEKE) 20(4), 575–608 (2010)
28. Cossentino, M., Gaglio, S., Garro, A., Seidita, V.: Method fragments for agent design methodologies: from standardisation to research. International Journal of Agent Oriented Software Engineering 1(1), 91–121 (2007)
29. Osterweil, L.: Software processes are software too. In: 9th International Conference on Software Engineering (ICSE 1987), pp. 2–13. IEEE CS Press, Los Alamitos (1987)
30. Cossentino, M., Gaud, N., Hilaire, V., Galland, S., Koukam, A.: ASPECS: an agent-oriented software process for engineering complex systems. International Journal of Autonomous Agents and Multi-Agent Systems (IJAAMAS) 20(2), 260–304 (2010)
31. Cossentino, M., Gaglio, S., Seidita, V.: Adapting PASSI to support a goal oriented approach: a situational method engineering experiment. In: 5th European Workshop on Multi-Agent Systems, EUMAS 2007 (2007)
32. Gleizes, M.P., Camps, V., George, J.P., Capera, D.: Engineering systems which generate emergent functionalities. In: Weyns, D., Brueckner, S., Demazeau, Y.E. (eds.) EEMMAS 2007. LNCS (LNAI), vol. 5049, pp. 58–75. Springer, Heidelberg (2008)
33. Jorquera, T., Bonjean, N., Gleizes, M.P., Maurel, C., Migeon, F.: Combining methodologies fragments using self-organizing MAS. Technical Report IRIT/RR-2010-4-FR, IRIT (2010)
34. Jorquera, T., Maurel, C., Migeon, F., Gleizes, M.P., Bonjean, N., Bernon, C.: ADELFE fragmentation. Technical Report IRIT/RR-2009-26-FR, IRIT (2009)
35. Cossentino, M., Sabatucci, L., Seidita, V.: Method fragments from the PASSI process. Technical Report RT-ICAR-21-03, Istituto di Calcolo e Reti ad Alte Prestazioni - Consiglio Nazionale delle Ricerche (2006)
36. Debenham, J., Henderson-Sellers, B.: Designing agent-based process systems – extending the OPEN process framework. In: Intelligent Agent Software Engineering, pp. 160–190. Idea Group Publishing, USA (2003)
37. Henderson-Sellers, B., Debenham, J., Tran, Q.-N.N., Cossentino, M., Low, G.: Identification of reusable method fragments from the PASSI agent-oriented methodology. In: Kolp, M., Bresciani, P., Henderson-Sellers, B., Winikoff, M. (eds.) AOIS 2005. LNCS (LNAI), vol. 3529, pp. 95–110. Springer, Heidelberg (2006)

38. Henderson-Sellers, B., Giorgini, P., Bresciani, P.: Evaluating the potential for integrating the OPEN and Tropos metamodels. In: Al-Ani, B., Arabnia, H.R., Mun, Y. (eds.) International Conference on Software Engineering Research and Practice, SERP 2003, USA, June 23-26, vol. 2, pp. 992–995. CSREA Press, Las Vegas (2003)
39. ISO/IEC: Software Engineering — Metamodel for Development Methodologies. Fdis 24744 edn. (2006)
40. Gonzalez-Perez, C., Henderson-Sellers, B.: Metamodelling for Software Engineering. Wiley, Chichester (2008)
41. Gonzalez-Perez, C., Henderson-Sellers, B.: On the ease of extending a powertype-based methodology metamodel. In: 2nd Workshop on Metamodelling, WoMM 2006 (2006)
42. Rumbaugh, J.E.: Notation notes: Principles for choosing notation. JOOP 9(2), 11–14 (1996)
43. Tran, Q.-N.N., Henderson-Sellers, B., Debenham, J.: Incorporating the elements of the MASE methodology into agent OPEN. In: 6th International Conference on Enterprise Information Systems, ICEIS 2004 (2004)
44. Henderson-Sellers, B., Debenham, J., Tran, Q.-N.N.: Adding Agent-Oriented Concepts Derived from Gaia to Agent OPEN. In: Persson, A., Stirna, J. (eds.) CAiSE 2004. LNCS, vol. 3084, pp. 98–111. Springer, Heidelberg (2004)
45. Henderson-Sellers, B., Tran, Q.-N.N., Debenham, J.: Incorporating Elements from the Prometheus Agent-Oriented Methodology in the OPEN Process Framework. In: Bresciani, P., Giorgini, P., Henderson-Sellers, B., Low, G., Winikoff, M. (eds.) AOIS 2004. LNCS (LNAI), vol. 3508, pp. 140–156. Springer, Heidelberg (2005)
46. Seidita, V., Cossentino, M., Gaglio, S.: A repository of fragments for agent systems design. In: Workshop From Objects To Agents, WOA 2006 (2006)
47. Cossentino, M., Sabatucci, L., Seidita, V.: A collaborative tool for designing and enacting design processes. In: Shin, S.Y., Ossowski, S., Menezes, R., Viroli, M. (eds.) 24th Annual ACM Symposium on Applied Computing (SAC 2009), Honolulu, Hawai'i, USA, vol. 2, pp. 715–721. ACM, New York (2009)
48. Bernon, C., Cossentino, M., Gleizes, M.P., Turci, P., Zambonelli, F.: A study of some multi-agent meta-models. In: Odell, J.J., Giorgini, P., Müller, J.P. (eds.) AOSE 2004. LNCS, vol. 3382, pp. 62–77. Springer, Heidelberg (2005)
49. Gonzalez-Perez, C., McBride, T., Henderson-Sellers, B.: A metamodel for assessable software development methodologies. Software Quality Journal 13(2), 195–214 (2005)
50. Henderson-Sellers, B., Gonzalez-Perez, C.: A comparison of four process metamodels and the creation of a new generic standard. Information & Software Technology 47(1), 49–65 (2005)
51. Molesini, A., Denti, E., Omicini, A.: From AO methodologies to MAS infrastructures: The SODA case study. In: Artikis, A., O'Hare, G.M.P., Stathis, K., Vouros, G.A. (eds.) ESAW 2007. LNCS (LNAI), vol. 4995, pp. 300–317. Springer, Heidelberg (2008)
52. Molesini, A., Denti, E., Omicini, A.: MAS meta-models on test: UML vs. OPM in the SODA case study. In: [83], pp. 163–172
53. Grupo de Investigación en Agentes Software: Ingeniería y Aplicaciones (2009), Home page, http://grasia.fdi.ucm.es/ingenias/metamodel/
54. Beydoun, G., Low, G.C., Henderson-Sellers, B., Mouratidis, H., Gómez-Sanz, J.J., Pavón, J., Gonzalez-Perez, C.: FAML: A generic metamodel for MAS development. IEEE Transactions on Software Engineering 35(6), 841–863 (2009)

55. Dam, K.H., Winikoff, M., Padgham, L.: An agent-oriented approach to change propagation in software evolution. In: Australian Software Engineering Conference, pp. 309–318. IEEE Computer Society, Los Alamitos (2006)
56. Giorgini, P., Mylopoulos, J., Perini, A., Susi, A.: The Tropos metamodel and its use. Informatica 29, 401–408 (2005)
57. Garcia-Ojeda, J.C., DeLoach, S.A., Robby, Oyenan, W.H., Valenzuela, J.L.: O-maSE: A customizable approach to developing multiagent development processes. In: Luck, M., Padgham, L. (eds.) Agent-Oriented Software Engineering VIII. LNCS, vol. 4951, pp. 1–15. Springer, Heidelberg (2008)
58. Cossentino, M., Gaglio, S., Sabatucci, L., Seidita, V.: The PASSI and Agile PASSI MAS meta-models compared with a unifying proposal. In: [83], pp. 183–192
59. Beydoun, G., Gonzalez-Perez, C., Low, G., Henderson-Sellers, B.: Synthesis of a generic MAS metamodel. In: 4th International Workshop on Software Engineering for Large-scale Multi-Agent Systems (SELMAS 2005), pp. 1–5. ACM, New York (2005)
60. Henderson-Sellers, B., Gonzalez-Perez, C.: Standardizing methodology metamodelling and notation: An iso exemplar. In: Kaschek, R., Kop, C., Steinberger, C., Fliedl, G. (eds.) UNISCON. Lecture Notes in Business Information Processing, vol. 5, pp. 1–12. Springer, Heidelberg (2008)
61. Bernon, C., Camps, V., Gleizes, M.P., Picard, G.: Engineering adaptive multi-agent systems: the ADELFE methodology. In: Agent Oriented Methodologies, pp. 172–202. Idea Group Publishing, USA (2005)
62. Zambonelli, F., Jennings, N.R., Wooldridge, M.: Developing multiagent systems: The Gaia methodology. ACM Transactions on Software Engineering and Methodology (TOSEM) 12(3), 317–370 (2003)
63. Pavòn, J., Gòmez-Sanz, J.J., Fuentes, R.: The INGENIAS methodology and tools. In: [84], ch. IX, pp. 236–276
64. DeLoach, S.A., Kumar, M.: Multi-agent systems engineering: An overview and case study. In: [84], ch. XI, pp. 317–340
65. Cossentino, M.: From requirements to code with the PASSI methodology. In: [84], ch. IV, pp. 79–106
66. Padgham, L., Winikoff, M.: Prometheus: A methodology for developing intelligent agents. In: Giunchiglia, F., Odell, J.J., Weiss, G. (eds.) AOSE 2002. LNCS, vol. 2585, pp. 174–185. Springer, Heidelberg (2003)
67. Omicini, A.: SODA: Societies and infrastructures in the analysis and design of agent-based systems. In: Ciancarini, P., Wooldridge, M.J. (eds.) AOSE 2000. LNCS, vol. 1957, pp. 185–193. Springer, Heidelberg (2001)
68. Molesini, A., Zhang, S.-W., Denti, E., Ricci, A.: SODA: A roadmap to artefacts. In: Dikenelli, O., Gleizes, M.-P., Ricci, A. (eds.) ESAW 2005. LNCS (LNAI), vol. 3963, pp. 49–62. Springer, Heidelberg (2006)
69. Cernuzzi, L., Rossi, G., Plata, L.: On the evaluation of agent oriented modeling methods. In: Workshop on Agent Oriented Methodology, pp. 21–30 (2002)
70. Bernon, C., Gleizes, M.P., Picard, G., Glize, P.: The ADELFE methodology for an intranet system design. In: Giorgini, P., Lespérance, Y., Wagner, G., Yu, E. (eds.) 4th International Bi-Conference Workshop on Agent-Oriented Information Systems (AOIS 2002), Toronto, Canada. CAiSE 2002, CEUR Workshop Proceedings, vol. 57 (2002)
71. Sturm, A., Shehory, O.: A framework for evaluating agent-oriented methodologies. In: Giorgini, P., Henderson-Sellers, B., Winikoff, M. (eds.) AOIS 2003. LNCS (LNAI), vol. 3030, pp. 94–109. Springer, Heidelberg (2004)

72. Dam, K.H.: Comparing agent-oriented methodologies. In: Giorgini, P., Henderson-Sellers, B., Winikoff, M. (eds.) AOIS 2003. LNCS (LNAI), vol. 3030, pp. 78–93. Springer, Heidelberg (2004)
73. Luck, M., Ashri, R., D'Inverno, M.: Agent-Based Software Development. Artech House, Norwood (2004)
74. Tran, Q.N.N., Low, G.C.: Comparison of ten agent-oriented methodologies. In: Agent Oriented Methodologies, pp. 341–367. Idea Group Publishing, USA (2005)
75. Padgham, L., Winikoff, M., DeLoach, S., Cossentino, M.: A unified graphical notation for AOSE. In: Luck, M., Gomez-Sanz, J.J. (eds.) AOSE 2008. LNCS, vol. 5386, pp. 116–130. Springer, Heidelberg (2009)
76. Schmidt, D.C.: Model-driven engineering. Computer 39(2), 25–31 (2006)
77. Miles, S., Groth, P.T., Munroe, S., Luck, M., Moreau, L.: AgentPrIMe: Adapting MAS designs to build confidence. In: Luck, M., Padgham, L. (eds.) Agent-Oriented Software Engineering VIII. LNCS, vol. 4951, Springer, Heidelberg (2008)
78. Spanoudakis, N., Moraitis, P.: Development with ASEME. In: 11th International Workshop on 11th International Workshop on Agent Oriented Software Engineering, AOSE (2010)
79. Wagner, G.: Agent-oriented analysis and design of organizational information system. In: 4th IEEE International Baltic Workshop on Databases and Information Systems (2000)
80. Argente, E., Botti, V., Vincente, J.: GORMAS: An organizational-oriented methodological guideline for open MAS. In: Proc. of Agent-Oriented Software Engineering (AOSE) Workshop (2009)
81. Garijo, F.J., Gòmez-Sanz, J.J., Massonet, P.: The MESSAGE methodoly for agent-oriented analysis and design. In: [84], ch. VIII, pp. 203–235
82. Castro, J., Kolp, M., Mylopoulos, J.: A requirements-driven development methodology. In: Dittrich, K.R., Geppert, A., Norrie, M.C. (eds.) CAiSE 2001. LNCS, vol. 2068, pp. 108–123. Springer, Heidelberg (2001)
83. Pěchouček, M., Petta, P., Varga, L.Z. (eds.): CEEMAS 2005. LNCS (LNAI), vol. 3690. Springer, Heidelberg (2005)
84. Henderson-Sellers, B., Giorgini, P.: Agent Oriented Methodologies. Idea Group Publishing, Hershey (2005)

Formal Methods in Agent-Oriented Software Engineering

Amal El Fallah-Seghrouchni[1], Jorge J. Gomez-Sanz[2], and Munindar P. Singh[3]

[1] LIP6 - University Pierre and Marie Curie,
104, Avenue du Président Kennedy, 75016, Paris, France
Amal.Elfallah@lip6.fr
[2] GRASIA Research Group, Universidad Complutense de Madrid
Avda. Complutense, 28040 Madrid, Spain
jjgomez@fdi.ucm.es
[3] Department of Computer Science, North Carolina State University,
Raleigh, NC 27695-8206, USA
singh@ncsu.edu

Abstract. There is a growing interest among agent and multiagent system developers for formal methods. Formal methods are means to define and realize correct specifications of multiagent system. The benefits of formal methods become clearer when we recognize the cost of developing a defective multiagent system. This paper seeks to introduce engineers to the possibilities of applying formal methods for multiagent systems. To this end, it discusses selected formal methods approaches for multiagent systems for which there is tool support. These works have been organized into two broad categories: those formal methods constituting a development method in themselves and those intended to complement an existing development method.

Keywords: Formal methods, AOSE, software engineering, specification, verification.

1 Introduction

Agent-Oriented Software Engineering (AOSE) borrows software engineering concepts and techniques with the intention of enabling a controlled development of multiagent systems (MAS). This fits software engineering goals, since software engineering is about making software development less an art and more a discipline. It requires, among others, defining engineering activities to be structured using a method. According to the Software Engineering Book of Knowledge [38], a method imposes structure on a software engineering activity to make the activity more systematic and therefore presumably more likely to succeed. A method may be heuristic (structured, data-oriented, or object-oriented); prototyping-based; or formal.

Literature in AOSE has mainly emphasized the heuristic and prototyping-based method approaches, which can be found in reviews of agent oriented

M.-P. Gleizes and J.J. Gomez-Sanz (Eds.): AOSE 2009, LNCS 6038, pp. 213–228, 2011.

methodologies such as Henderson-Sellers and Giorgini [36] or Bergenti et al. [7]. This survey focuses on the third category, namely, to formal methods. A formal method refers to the use of mathematically based methods which are typically applied in form of specification language or notations; the transformation of specifications; or verification [38].

Since our context is software engineering, this survey emphasizes those works illustrating the contribution of a formal method to software development. Concretely, this survey addresses those formal method solutions that serve a well-defined, discrete function in AOSE and which are assisted by available tools. Because existing AOSE approaches address the whole gamut of the stages of software development, formal methods can be applied to these stages as well, including analysis, design, implementation, testing, validation, and verification. Formal methods also apply during the main phases of the software life cycle: downstream, upstream, and during deployment.

Taking these ideas into account, we organize the paper as follows. Section 2 describes works that emphasize the formal representation and tooling aspects of formal methods. This enables us to reuse the techniques in an existing AOSE methodology. Section 3 moves up the food chain and describes works that emphasize the methodological aspects of formal methods, i.e., the way from a specification to the desired multiagent system. Section 4 provides additional references to other works which can be used to complement this survey. Finally, Section 5 concludes with a summary.

2 Formal Specification and Tools

Formal specification involves the use of a formal notation to describe the system under development. A formal specification can be as precise and complex as to resemble an executable program, but that is not typically valuable. Specifying is not programming, though it may be the case some specifications can be interpreted and executed. Hence, we do not consider agent-oriented programming languages unless they serve for specification purposes rather than implementation. As a consequence, we emphasize works carrying out formal derivation or verification on the resulting specification and always with the assistance of available software tools.

2.1 Formal Specification

The formal representation methods listed here fall into four main themes: (1) first-order logic and one of its variants, the situation calculus; (2) temporal logics; (3) process algebras; and (4) automata.

The first theme includes the SMART framework from D'Inverno and Luck [24] and Cognitive Agents Specification Language (CASL) from [56]. The SMART framework [24] is formalized using the Z specification language, and supports the concepts of entities, objects, agents (objects with goals), and autonomous agents (agents with motivations). The SMART framework takes advantage of

various Z tools to perform simulations and verify properties. A MAS specified with SMART is built using the actSMART agent implementation [6], which specializes AgentSpeak(L) to adapt to the principles of SMART. Hilaire et al. [37] object that Z cannot be used to address reactive aspects, and propose combining statecharts (an extension of finite automata to support parallelism, nested states, and broadcast communication for events) and Object-Z to provide MAS specifications. The combination of statecharts and Object-Z is a multiformalism specification called OZS. ForMAAD [34], reviewed in Section 3, offers a hybrid approach combining Z and temporal logic that enables us to talk about dynamic aspects of the MAS constituents.

Situation calculus gives first-class status to situations and captures the intuition of transitions between situations. Cognitive Agents Specification Language (CASL) [56] [57], based on the situation calculus, is an extension of the well-known ConGolog. CASL provides a verification environment based on the Prototype Verification System (PVS), called CASLve. CASL can express what the mental states of the agents, their goals, and the effects of their actions.

In the second theme, temporal logics explicitly support the concept of time. Fisher [26] reviews different implementations and techniques in the context of software agents. Several works apply temporal logics as the basis for specification languages. This is the case of Formal Tropos [47] [29] and ForMAAD [34].

Process algebras have been studied as well as formalisms for specifying MAS. Though applied to formalize different aspects of existing MAS, as the formalization of Cougaar agent architecture [33], there is a gap in the literature in this respect [63]. Despite the gap, it is possible to find works that, rather than declaring the semantics of an existing agent framework or language, provide an intuitive orientation as specification languages. Specific examples are the API-calculus [48] and CSP‖B [54].

The API-calculus, from Rahimi et al. [48], is an algebra for agents that addresses mobility, security, natural grouping, and intelligence. Visualization of API-calculus specifications can be made with the ACVisualizer [3]. Some of these ideas have crystallized into the CAML process algebra [4], which regards each agent as a tuple of knowledge, conditions, capabilities, an event queue, plan, and an intention queue.

CSP‖B [54] is a promising specification method that can be used to specify multiagent systems, as Colin et al. [18] show for a vehicle coordination problem. In CSP‖B, each agent is a process and it can be made of either other processes or B machines. Hence, communication is expressed mainly with CSP, while the internal apparatus of each agent combines CSP and the B notation. B machines require the definition of invariants (properties that always hold) and operations that declare how the state is modified. CSP requires us to express communication channels and the expected behavior depending on the kind of information arriving from a channel. This notation is supported by two tools: for CSP support, FDR2 (`http://www.fsel.com/`) and for the B method, B4FREE `http://www.b4free.com`

Aside from the algebraic and logic-based approaches, we consider transition systems and automata-based approaches. Such approaches have been used to describe dynamic aspects within MAS, especially communications. Automata are of many varieties. Colored Petri Nets (CPNs) [44] are used to capture information coming from AUML protocol diagrams. CPNs can be analyzed with CPN tools (`http://wiki.daimi.au.dk/cpntools/cpntools.wiki`). The internals of the agents can also be described with automata, as illustrated by Fallah-Seghrouchni and colleagues [25]. They use hybrid automata (made of an automaton and a set of variables associated with the transitions of the automaton) to represent a multiagent planning framework that has been applied in an industrial context, namely, tactical aircraft simulation at Dassault-Aviation in France. The multiagent plan is a synchronized network of *hybrid automata*, each automaton representing an individual plan of an agent. Several mechanisms of control and validation are proposed and base on HyTech (`http://embedded.eecs.berkeley.edu/research/hytech/`) model checker to verify properties.

An automata-based representation is applied in the Interpreted Systems Programming Language (ISPL). ISPL is a language to define MAS used as input for the MCMAS [42], a model checker tool. ISPL can capture a variety of modeling concepts including time, knowledge, and obligations. The description resembles an automaton, since it involves the declaration of initial states and the evolution of the variables defining the state.

To conclude, automata can be combined with other formalisms, as shown by Hilaire et al. [37], who combine statecharts with Z notation to specify systems. The specification can be then processed using the STATEMATE [35] tool to perform prototyping and simulation.

2.2 Tools for Verification

One of the advantages of having a formal specification is being able to determine whether the specification satisfies some desired properties. In most cases, the desirable properties of the system are determined during the system specification, where initial requirements are identified. Examples of such properties can be found in Brazier [13] and Singh [58]. There are few generic properties, and most of them refer to interacting protocols and the livelocks or deadlocks among the agents. In general, it is the developer, using as starting point the requirements of the system, who determines what properties must be satisfied. Once a property is identified, it must be expressed using a suitable formalism, e.g., using temporal or epistemic logics.

Tools for verification require the object of the analysis to be declared using a tool-specific language. The degree of compatibility between this language and the one used for the specification varies. Sometimes, the specification itself can be supplied as is to the tool. In other cases, a translation is required, forcing us to ensure the correctness of the translation and that a property holding on the translation holds in the original specification as well [17].

In general, verifying the whole specification can be expensive. This forces us to either focus on different aspects or parts of the system and to progress incrementally. In this case, if a property is verified in a piece of the specification, it is desirable that the property remains when other pieces of the specification are included, or when the specification is refined. Some authors talk about the compositionality of properties, something which has been specially explored in the DESIRE framework [14]. DESIRE applies compositional verification, where the properties of the system as a whole are derived from properties of the corresponding agents, and the properties of each agent are derived from its constituting components. DESIRE comes with a software environment that has been validated via various projects.

Finally, when verifying properties in software, there is an important reminder: some properties, such as the halting problem for Turing machines, are undecidable; others may be decidable but intractable. With the above ideas in mind, we can understand the possibilities and disadvantages of the tools supporting verification of properties in system specifications. The study of existing works in the agent literature has motivated a division of tools into two major groups: model checking and theorem proving tools.

Model Checking. Model checking is about verifying if a specified logical formula is true in a specified model. Usually the formula is in temporal logic and the model is presented via a finite automaton. Temporal logic formulas help express properties that never hold, always hold, or are satisfied in some future states. The model represents the possible evolutions of the states of a software system such as a MAS. When representing a software system, the state space to explore can become too large to be tractable. Current model checking works rely on the capability of the model checker to reduce the state search space, though, sometimes, it is required to devise new techniques [10]. As mentioned earlier, another alternative consists in not translating the whole specification but a part, trying to focus on specific aspects of the system whose analysis requires less computation power. It happens that the system under study may require an infinite search space. Those cases will require applying bounded model checking or refer to advanced model checking works dealing with unbounded systems [45].

Existing works on model checking and MAS specifications use diverse tools. Several works use Spin, which is a linear-time model checker, meaning that it considers individual runs or computation paths (`http://spinroot.com/`). Other works use the JavaPathFinder tool, which can be downloaded from `http://babelfish.arc.nasa.gov/trac/jpf`. The uses of these tools and the kinds of properties each work considers are reviewed below.

Lacey and DeLoach [40] illustrate the application of the Spin model checking tool to MAS specifications created with agenTool. Interactions are expressed in agenTool using UML-like statecharts. Lacey and DeLoach introduce an algorithm that converts statecharts into Promela code, which is the input required by Spin. Lacey and DeLoach use sequence diagrams to describe expected sequences of messages. General properties verified with Spin are deadlocks (conversation ends in other state different from one of the end states defined in the specification),

liveness (looking for occurrences of all states of the conversation at least in one execution path), and unused states or messages.

MABLE [65] is a programming language for agents enriched with constructs for verification. It takes advantage of Spin by translating a MABLE program into Promela. Properties regard the compliance with the semantics of communicative acts. This implies verifying the preconditions and postconditions of each communicative act are met.

Desai et al. [23] address the problem of verifying commitment-based interaction protocols by mapping commitments and protocols to Promela representations for verification using Spin. A protocol enactment is deemed noncompliant if it terminates in a state where an unconditional commitment is still active. Also, upon completion, there should be no pending messages to process. There are specific properties that may need to be defined, like, in a buyer-seller scenario, if a buyer rejects a quote, no goods should be shipped.

Dennis et al. [21] introduce a generic framework, the Agent Infrastructure Layer AIL, aiming to check not only what an agent does, but also its intentions. AIL captures the commonalities among some programming languages [22]. The idea is to capture the essence of a representative collection of programming languages for agents, specifically, AgentSpeak, 3APL, Jadex, and MetateM. The result would be transformed into a Java program, which then would be run using the Agent Java PathFinder, an extension of Java PathFinder. This extension provides additional state reduction techniques [10]. In the example, authors illustrate the verification of postconditions of plan execution.

Other hybrid approaches combine different tools. Spin and Java PathFinder are used in [11] as part of the CASP (Checking AgentSpeak Programs) toolkit [9]. The work assumes a multiagent system developed using AgentSpeak$F$, a variant of AgentSpeak$L$ [49] that represents agents using finite state automata. This program is translated into Promela to be used as input to Spin. Also, this work shows how Java PathFinder model checking tools can be applied to AgentSpeak when the same AgentSpeak code is translated into Java. Verified properties regard the beliefs of agents and their intentions along any possible execution. These properties are specific to the case studies handled by authors and are not intuitively reusable.

Hilaire et al. [37], in their RIO framework, apply Harels [35] STATEMATE environment for the prototyping and the simulation of statecharts to conduct verification and prototyping. STATEMATE permits the simulation and prototyping of the statechart specifications. Hilaire et al. report as well to have used SAL 2 [20], which provides high performance symbolic and bounded model checkers. Hilaire et al. suggest combining bounded model checking with inductive theorem proving to deal with combinatorial explosion of states, though no specific software tools for this are mentioned. Verified properties are particular to the problem used as case study: two robots having to explore a narrow corridor and how they coordinate to let the other pass. In this case study, it is shown that the least constrained robot becomes altruist.

Some works with Hybrid automata [25] report to have used HyTech. The properties addressed include task covering by an automaton, the availability of resources, and multiagent plan feasibility.

NuSMV verifies formulas expressed in the well-known branching-time temporal logic called Computation Tree Logic (CTL) [16], which enables expressing properties dealing with all or some future computations (branches) of a software system. Telang and Singh [59] employ NuSMV as a basis for verifying whether the sequence diagrams corresponding to a system realization satisfy the commitments included in a business model.

Lomuscio et al. [42] have created MCMAS, a model checker tool which specifically deals with agent-based specifications and scenarios. The MCMAS input description language is called ISPL and was introduced above. Properties verified in the case studies are not reusable, but are expressed using CTL and epistemic operators, operators for reasoning about the actions of the agents, and strategies. Hence, in MCMAS, it is possible to make references to what each agent know or does not know along the time. MCMAS admits as well fairness formulae (to be assumed as always true), and definition of the number of agents to be used in the proof.

Recently, Gerard and Singh have developed a tool called Proton for reasoning about protocol specifications [30]. Proton addresses the problem of whether a protocol refines another, taking into account differences in roles and the specific interactions performed in each protocol. For example, we can think of Pay as a protocol by which a payer pays a payee, thereby discharging a suitable commitment. Intuitively, the protocol PayByCheck accomplishes payment as well and should be considered a refinement of Pay even though it involves an additional role (the bank) and involves entirely different messages from Pay. Proton takes two protocol specifications along with a conceptual mapping from one protocol to another and generates suitable ISPL and CTL specifications, which when run through MCMAS verify if the refinement holds.

Riemsdijk et al. [50] propose BUpL, the Belief Update programming Language implemented with Maude. BUpL programs can be executed, tested, and verified. Verification is carried out using the LTL model checker from Maude. The properties are expressed in Linear Temporal Logic and the developer must identify the elements from the program which represents the states of the system, and the atomic predicates that can evaluated in those states. Properties that can be verified mainly involve the reachability of the identified states. For instance, that a goal should be reached and that the agent never comes to believe some information.

Automated Theorem Proving. Automated theorem proving consists in, from a set of theories and axioms, affirming the validity of a formula. In the agent literature, it has been applied thoroughly in works related with computational logic, such as MetateM [28] and ConGolog [31]. Nevertheless, since their use requires more training than the model checking alternative, it is harder to find proposals where theorem proving tools take an active part in the development. As an example, four works are listed below.

The Prototype Verification System (PVS) is a theorem prover that takes as input higher-order logic predicates and can be downloaded from `http://pvs.csl.sri.com/`. It is used in the construction of the CASLve, which supports the verification of CASL specifications [55] [41]. The verified properties concern the desirable properties of the specification language necessary to progress in the verification effort. Aside from these, Shapiro and colleagues provide examples of problem specific properties related to the problem requirements, for example that in a meeting scheduler, the participants sit together in the meeting table at some point.

DESIRE [12] uses the Temporal Traces Language where the dynamics of the system are represented as an evolution of states of agents and an environment over time. It can be used to specify the system and observe its behavior prior to execution. In general, approaches using temporal logics can take advantage on existing theorem provers for temporal logic [26].

SOCS-SI is a tool that uses the SCIFF Proof [15] procedure to check the compliance of an agent interaction with respect to a protocol declaration. It is available from `http://lia.deis.unibo.it/sciff`. SCIFF is an abductive proof method and is implemented using Prolog. SCIFF does the verification using partial knowledge, something possible due to the abductive nature of the proof. It does not declare a state machine, but indicates which actions are forbidden, required, and possible.

Alechina et al. [5] propose a simplified version of 3APL, called SimpleAPL, where liveness and safety properties can be proven. Programs are translated into Propositional Dynamic Logic (PDL). The properties to verify are written in PDL as well, and then proven by means of PDL theorem provers. Alechina et al. use MSPASS (available from `http://www.cs.man.ac.uk/~schmidt/mspass` which has been incorporated into SPASS `http://project.kjsmith.net/`) and PDL-Tableau (`http://www.cs.man.ac.uk/~schmidt/pdl-tableau/`). The properties verified deal with the expected state of the system, including its agents, at some point.

3 Developing with Formal Methods

The formal development of MAS involves the definition of a methodology where formal methods play a dominant role. Section 2 introduced several formal notations and verification tools. Nevertheless, development itself, i.e., the construction of a MAS has not been addressed yet. In this section, we introduce three alternatives: formal derivation, integration with an existing methodology, or proposing a new one.

Formal derivation is a technique wherein one would automatically realize a system based on a given specification. Derivation can be understood as a form of model-to-code transformations, although more systematic and less based on the subjective judgment of a developer. An important approach is due to Vasconcelos [61] [60], where a multiagent system, coded using logic programming, is inferred from a specification, and then transformed using successive refinements into a

system adapted to the user's needs. However, by and large, formal derivation is conceptually and technically complex. As a result, formal methods usually involve a significant component of modeling methodologies.

The enhancement of existing development methods is easier to find. Three important illustrations of enhancement are Formal Tropos, Gaia, and MaSE. Formal Tropos [47] [29] combines temporal constraints into the specification of early requirements. This extension is supported by the T-Tool (downloadable from `http://www.dit.unitn.it/~ft/ft_tool.html`), which is based on the symbolic model checking tool NuSMV. NuSMV verifies formulas expressed in the well-known branching-time temporal logic called Computation Tree Logic (CTL), which enables expressing properties dealing with all or some future computations (branches) of a software system. Telang and Singh [59] employ NuSMV as a basis for verifying whether the sequence diagrams corresponding to a system realization satisfy the commitments included in a business model.

Gaia [66] was an early attempt to integrate formal methods into agent-oriented software engineering. Although its scope is limited to analysis, Gaia combined describing properties and modeling. Its major drawback was the lack of support tools, something addressed in part with the MADSK platform [32], which supports verifying liveness properties.

MaSE verifies interaction protocols using Spin model checking tool, as seen in Lacey and DeLoach [40]. Interactions are expressed in agenTool using UML-like statecharts. Lacey and DeLoach introduce an algorithm that converts statecharts into Promela code, which is the input required for the Spin model checker. Expected sequences of messages (obtained from current sequence diagrams), deadlocks (if a conversation ends in other state different from one of the end states defined in the specification), liveness (looking for occurrences of all states of the conversation at least in one execution path), and unused states or unused messages are detected. There are other errors that agentTool and Spin cannot detect, like timing errors (messages taking too much time to be sent or received, for instance), hardware failures, guard conditions of the sequences, interacting conversations causing a deadlock,.

Instead of enhancing an existing methodology, some authors propose new approaches for the creation of MAS. This is the case of the two following works: CASL/ConGolog [41] and ForMAAD [34]. The combination of CASL and ConGolog with i* leads to a formal development method [41]. Given a set of i* diagrams that capture the system requirements, ConGolog can formally capture information about actors, tasks, processes, and goals, using as its starting point information from i* Strategic Rationale (SR) diagrams. *Annotated SR diagrams* include additional information, which enable an executable and observable ConGolog program. Also, tests can be applied to an ongoing simulation to verify if specified properties hold. *Intentional Annotated SR* diagrams capture the knowledge and intentions of the agents. Such diagrams are transformed into CASL, thereby enabling the CASLve tool to be used to verify properties, more thoroughly than with ConGolog.

ForMAAD, from Hadj-Kacem et al. [34], is a formal methodology based in a combination of Z and temporal logics. It proposes a set of activities where the specification is further refined. Each refinement is analyzed to check it still satisfies the initial requirements. This proof is made with the aid of the tool Z/EVES [52].

4 Related Work

In an earlier survey, Wooldridge [64] classified AOSE formal methods into three areas: *formal modeling* (dealing with the use of formal methods to specify a system); *formal derivation* (dealing with generating a MAS from a given specification); and *verification* (dealing with the application of formal methods to the verification of properties in MAS). We find that Wooldridge's classification, although intuitive, no longer holds up in the literature. The first reason is the low number of works addressing formal derivation. Current research is more focused on model transformations, as seen in Nunes et al. [46]. The second reason is the availability of software tools for proving the satisfaction of properties. These tools enable us to focus more on the issue of determining what has to be proven and how to better use the available tools to do so. Addressing these problems requires some mathematical skills but is mainly a question of engineering. The above reasons have led us to pay attention to the method in the classification of works in this survey and to provide hints along the survey on the way verification is applied.

Other surveys exist that defend alternative perspectives on the application of formal methods to MAS development [19,51]. Our present survey seeks to motivate developers to apply formal methods, and so, we have filtered out works that did not promote any software development tool. Dastani et al. [19] and Rouff et al. [51] do not apply this restriction and, hence, include a wider set of works. Agent-oriented programming languages with formal basis have not been addressed here, except in cases where there was interest in the support tools. The reason has been our focus on the formal method approach not on the concrete implementation. Thus, if a language was introduced as a programming language and not as a specification language, such works were not included. Nevertheless, readers interested in programming languages based on computational logic can review the following works: Sadri and Toni [53] and Bordini et al. [8]. Sadri and Toni [53] categorize computational logic based approaches attending to how they contribute to the construction of multiagent systems, e.g., knowledge representation or realization of the observe-think-act cycle inside an agent. A complement to the present survey is the survey by Mascardi et al. [43], which discusses BDI implementations based on computational logic. Fisher et al. [27] present another survey of works related to computational logic. Since some approaches to the implementation of MAS are not based on logics, it is worth reviewing Bordini et al.s survey [8], which accounts for programming languages

and framework, classifying them according to various programming paradigms, including the imperative and object-oriented paradigms.

Several scientific events emphasize formal methods for multiagent systems. These include the *Autonomous Agents and MultiAgent Systems* (AAMAS) conference and workshops on *Agent Oriented Software Engineering*, *Declarative Agent Languages and Technologies*, *From Agent Theory to Agent Implementation* (AT2AI), and *International Workshop on Computational Logic in MultiAgent Systems* (CLIMA).

5 Conclusions

The history of the development of formal methods for multiagent systems mirrors the history of multiagent systems broadly. The earliest formal methods sought to apply well-known formal techniques from traditional software engineering, such as temporal logic and Petri nets. These methods, exemplified by [14,24,33], generally did not emphasize high-level abstractions specific to multiagent systems, treating them mostly as conventional distributed systems. The next broad phase of research brought in high-level abstractions that are central to different theories of agents and multiagent systems, such as those involving beliefs and intentions, commitments, and communications. These methods, exemplified by [56], tended to be custom methods. Researchers realized that the effort required in tooling is substantial. This has led to the modern research on formal methods, exemplified by [11], wherein multiagent abstractions are mapped to the representations of existing tools so as to leverage them for verifying MAS. There are notable exceptions such as MCMAS [39] and SOCS-SI [15], which are extensive tools that are customized to common agent abstractions.

Automatic theorem proving approaches seem to have had less impact than some predicted. Model checking is the predominant approach. A major reason may be the availability of tools for model checking and the apparent naturalness with respect to which models can be specified. Most often, the formalisms used in MAS community are logic-based (e.g. temporal logic).

When we recognize that multiagent systems are often complex software systems, we can see that the need for robust methodologies including formal methods applies in multiple points in their life cycle. MAS should be verified not only in the design phase, but also during implementation, and even during deployment (MAS should be endowed with monitoring tools). In particular, when a multiagent system consists of autonomous, heterogeneous agents, there is no well-defined central authority that can examine the construction or run-time state of a given agent. In such cases, we must back away from verification in the traditional sense outlined above and consider ways to check if the behavior of an agent is compliant with respect to high-level requirements, such as those expressed via commitments [62]. Possible approaches would involve having each agent or maintain logs regarding its interactions with other agents. Verification on the fly could also be suitable for MAS systems.

Formal methods for MAS garner a steady, though arguably limited, amount of attention from the research community. Each year, we see new results showing approaches that apply formal methods to AOSE, typically on specific case studies. We hope that as formal methods mature, tools for them will improve, leading to a greater interest in incorporating them in all stages of the life cycle of the development and deployment of robust multiagent systems.

Acknowledgements

Jorge J. Gomez-Sanz has partially been funded by the the project Agent-based Modelling and Simulation of Complex Social Systems (SiCoSSys), supported by Spanish Council for Science and Innovation, with grant TIN2008-06464-C03-01, and by the Programa de Creación y Consolidación de Grupos de Investigación UCM-Banco Santander for the group number 921354 (GRASIA group).

Munindar Singh's effort was partially supported by National Science Foundation Grant #0910868.

References

1. The First International Joint Conference on Autonomous Agents & Multiagent Systems, Bologna, Italy, Proceedings. ACM, New York (2002)
2. The Third International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2004). IEEE Computer Society, New York (2004)
3. Ahmad, R., Rahimi, S.: ACVisualizer: A visualization tool for API-calculus. Multiagent and Grid Systems 4(3), 271–291 (2008)
4. Ahmad, R., Rahimi, S., Gupta, B.: An Intelligence-Aware Process Calculus for Multi-Agent System Modeling. In: International Conference on Integration of Knowledge Intensive Multi-Agent Systems 2007, pp. 210–215 (2007)
5. Alechina, N., Dastani, M., Logan, B.S., Meyer, J.-J.C.: A logic of agent programs. In: Proceedings of the 22nd National Conference on Artificial Intelligence, pp. 795–800. AAAI Press, Menlo Park (2007)
6. Ashri, R., Luck, M., d'Inverno, M.: From SMART to agent systems development. Engineering Applications of Artificial Intelligence 18(2), 129–140 (2005)
7. Bergenti, F., Gleizes, M.-P., Zambonelli, F. (eds.): Methodologies and Software Engineering for Agent Systems. Kluwer Academic, Dordrecht (2004)
8. Bordini, R.H., Braubach, L., Dastani, M., El Fallah-Seghrouchni, A., Gómez-Sanz, J.J., Leite, J., O'Hare, G.M.P., Pokahr, A., Ricci, A.: A Survey of Programming Languages and Platforms for Multi-Agent Systems. Informatica 30(1), 33–44 (2006)
9. Bordini, R.H., Fisher, M., Pardavila, C., Visser, W., Wooldridge, M.: Model checking multi-agent programs with CASP. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 110–113. Springer, Heidelberg (2003)
10. Bordini, R.H., Fisher, M., Visser, W., Wooldridge, M.: State-Space Reduction Techniques in Agent Verification. In: AAMAS 2004 [2], pp. 896–903 (2004)
11. Bordini, R.H., Fisher, M., Visser, W., Wooldridge, M.: Verifying Multi-agent Programs by Model Checking. Autonomous Agents and Multi-Agent Systems 12(2), 239–256 (2006)

12. Bosse, T., Jonker, C.M., Meij, L., van der Sharpanskykh, A., Treur, J.: Specification and Verification of Dynamics in Cognitive Agent Models. In: Proceedings of the 6th IEEE/WIC/ACM International Conference on Intelligent Agent Technology, Washington, DC, USA, pp. 247–254. IEEE Computer Society, Los Alamitos (2006)
13. Brazier, F.M.T., Cornelissen, F., Gustavsson, R., Jonker, C.M., Lindeberg, O., Polak, B., Treur, J.: Compositional verification of a multi-agent system for one-to-many negotiation. Applied Intelligence 20(2), 95–117 (2004)
14. Brazier, F.M.T., Jonker, C.M., Treur, J.: Principles of component-based design of intelligent agents. Data Knowledge Engineering 41(1), 1–27 (2002)
15. Chesani, F., Gavanelli, M., Alberti, M., Lamma, E., Mello, P., Torroni, P.: Specification and verification of agent interaction using abductive reasoning (tutorial paper). In: Toni, F., Torroni, P. (eds.) CLIMA 2005. LNCS (LNAI), vol. 3900, pp. 243–264. Springer, Heidelberg (2006)
16. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge (1999)
17. Cohen, M., Dam, M., Lomuscio, A., Russo, F.: Abstraction in model checking multi-agent systems. In: The 8th International Joint Conference on Autonomous Agents and Multiagent Systems, pp. 945–952. IFAAMAS (2009)
18. Colin, S., Lanoix, A., Kouchnarenko, O., Souquières, J.: Using CSP||B components: Application to a platoon of vehicles. In: Cofer, D., Fantechi, A. (eds.) FMICS 2008. LNCS, vol. 5596, pp. 103–118. Springer, Heidelberg (2009)
19. Dastani, M., Hindriks, K.V., Meyer, J.-J.C. (eds.): Specification and Verification of Multi-agent Systems. Springer, US (2010)
20. de Moura, L.M., Owre, S., Rueß, H., Rushby, J.M., Shankar, N., Sorea, M., Tiwari, A.: SAL 2. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 496–500. Springer, Heidelberg (2004)
21. Dennis, L.A., Farwer, B., Bordini, R.H., Fisher, M.: A flexible framework for verifying agent programs. In: The 7th International Joint Conference on Autonomous Agents and Multiagent Systems, pp. 1303–1306. IFAAMAS (2008)
22. Dennis, L.A., Farwer, B., Bordini, R.H., Fisher, M., Wooldridge, M.: A Common Semantic Basis for BDI Languages. In: Dastani, M.M., El Fallah Seghrouchni, A., Ricci, A., Winikoff, M. (eds.) ProMAS 2007. LNCS (LNAI), vol. 4908, pp. 124–139. Springer, Heidelberg (2007)
23. Desai, N., Cheng, Z., Chopra, A.K., Singh, M.P.: Toward verification of commitment protocols and their compositions. In: The 6th International Joint Conference on Autonomous Agents and Multiagent Systems, pp. 33–35. IFAAMAS (2007)
24. D'Inverno, M., Luck, M.: Understanding Agent Systems. Springer, Heidelberg (2004)
25. Fallah-Seghrouchni, A.E., Degirmenciyan-Cartault, I., Marc, F.: Modelling, Control and Validation of Multi-agent Plans in Dynamic Context. In: AAMAS 2004 [25], pp. 44–51 (2004)
26. Fisher, M.: Implementing temporal logics: Tools for execution and proof (Tutorial paper). In: Toni, F., Torroni, P. (eds.) CLIMA 2005. LNCS (LNAI), vol. 3900, pp. 129–142. Springer, Heidelberg (2006)
27. Fisher, M., Bordini, R.H., Hirsch, B., Torroni, P.: Computational Logics and Agents: A Road Map of Current Technologies and Future Trends. Computational Intelligence 23(1), 61–91 (2007)
28. Fisher, M., Wooldridge, M.: On the Formal Specification and Verification of Multi-Agent Systems. International Journal of Cooperative Information Systems 6(1), 37–66 (1997)

29. Fuxman, A., Liu, L., Mylopoulos, J., Roveri, M., Traverso, P.: Specifying and analyzing early requirements in tropos. Requirements Engineering 9(2), 132–150 (2004)
30. Gerard, S.N., Singh, M.P.: Formalizing and Verifying Protocol Refinement. TR 18, North Carolina State University (2010)
31. De Giacomo, G., Lespérance, Y., Levesque, H.J.: ConGolog, a concurrent programming language based on the situation calculus. Artificial Intelligence 121(1-2), 109–169 (2000)
32. Gorodetsky, V., Karsaev, O., Samoylov, V., Konushy, V.: Support for analysis, design, and implementation stages with MASDK. In: Luck, M., Gomez-Sanz, J.J. (eds.) AOSE 2008. LNCS, vol. 5386, pp. 272–287. Springer, Heidelberg (2009)
33. Gračanin, D., Singh, H.L., Hinchey, M.G., Eltoweissy, M., Bohner, S.A.: A CSP-Based Agent Modeling Framework for the Cougaar Agent-Based Architecture. In: IEEE International Conference on the Engineering of Computer-Based Systems, pp. 255–262 (2005)
34. Hadj-Kacem, A., Regayeg, A., Jmaiel, M.: ForMAAD: A formal method for agent-based application design. Web Intelligence and Agent Systems 5(4), 435–454 (2007)
35. Harel, D., Lachover, H., Naamad, A., Pnueli, A., Politi, M., Sherman, R., Shtull-Trauring, A., Trakhtenbrot, M.: STATEMATE: a working environment for the development of complex reactive systems. IEEE Transactions on Software Engineering 16(4), 403–414 (1990)
36. Henderson-Sellers, B., Giorgini, P. (eds.): Agent-Oriented Methodologies. Idea Group Pub., Hershey (2005)
37. Hilaire, V., Gruer, P., Koukam, A., Simonin, O.: Formal driven prototyping approach for multiagent systems. International Journal of Agent-Oriented Software Engineering 2(2), 246–266 (2008)
38. IEEE Computer Society: Software Engineering Body of Knowledge (SWEBOK). Angela Burgess, EUA (2004)
39. Kacprzak, M., Lomuscio, A., Penczek, W.: Verification of Multiagent Systems via Unbounded Model Checking. In: AAMAS 2004 [2], pp. 638–645 (2004)
40. Lacey, T., DeLoach, S.A.: Automatic Verification of Multiagent Conversations. In: Proceedings of the 11th Annual Midwest Artificial Intelligence and Cognitive Science Conference, pp. 93–100 (2000)
41. Lapouchnian, A., Lespérance, Y.: Using the conGolog and CASL formal agent specification languages for the analysis, verification, and simulation of i* models. In: Borgida, A.T., Chaudhri, V.K., Giorgini, P., Yu, E.S. (eds.) Conceptual Modeling: Foundations and Applications. LNCS, vol. 5600, pp. 483–503. Springer, Heidelberg (2009)
42. Lomuscio, A., Qu, H., Raimondi, F.: MCMAS: A model checker for the verification of multi-agent systems. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 682–688. Springer, Heidelberg (2009)
43. Mascardi, V., Demergasso, D., Ancona, D.: Languages for Programming BDI-style Agents: an Overview. In: Workshop Dagli Oggetti agli Agenti, pp. 9–15. Pitagora Editrice Bologna (2005)
44. Mazouzi, H., Fallah-Seghrouchni, A.E., Haddad, S.: Open protocol design for complex interactions in multi-agent systems. In: AAMAS 2002 [1], pp. 517–526 (2002)
45. McMillan, K.L.: Methods for exploiting SAT solvers in unbounded model checking. In: the First International Conference on Formal Methods and Models for Co-Design, pp. 135–152. ACM/IEEE Computer Society (2003)

46. Nunes, I., Cirillo, E., de Lucena, C.J.P., Sudeikat, J., Hahn, C., Gomez-Sanz, J.J.: A survey on the implementation of agent oriented specifications. In: Gleizes, M.-P., Gomez-Sanz, J.J. (eds.) AOSE 2010. LNCS, vol. 6038, pp. 157–167. Springer, Heidelberg (2010)
47. Perini, A., Pistore, M., Roveri, M., Susi, A.: Agent-oriented modeling by interleaving formal and informal specification. In: Giorgini, P., Müller, J.P., Odell, J.J. (eds.) AOSE 2003. LNCS, vol. 2935, pp. 36–52. Springer, Heidelberg (2004)
48. Rahimi, S., Cobb, M., Ali, D., Petry, F.: A Modeling Tool for Intelligent-Agent Based Systems: the API-Calculus. In: Soft-Computing Agents: a new perspective for Dynamic Information Systems, pp. 165–186. IOS-Press, Amsterdam (2002)
49. Rao, A.S.: AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language. In: Perram, J., Van de Velde, W. (eds.) MAAMAW 1996. LNCS, vol. 1038, pp. 42–55. Springer, Heidelberg (1996)
50. Riemsdijk, M., Aştefănoaei, L., Boer, F.: Using the Maude Term Rewriting Language for Agent Development with Formal Foundations. In: Dastani, M., Hindriks, K.V., Meyer, J.-J.C. (eds.) Specification and Verification of Multi-agent Systems, pp. 255–287. Springer, Heidelberg (2010)
51. Rouff, C.A., Hinchey, M., Rash, J., Truszkowski, W., Gordon-Spears, D. (eds.): Agent Technology from a Formal Perspective. NASA Monographs in Systems and Software Engineering. Springer-Verlag London Limited, Heidelberg (2006)
52. Saaltink, M.: The Z/EVES System. In: Till, D., P. Bowen, J., Hinchey, M.G. (eds.) ZUM 1997. LNCS, vol. 1212, pp. 72–85. Springer, Heidelberg (1997)
53. Sadri, F., Toni, F.: Computational Logic and Multi-Agent Systems: a Roadmap. Computational Logic, Special Issue on the Future Technological Roadmap of Compulog-Net (1999)
54. Schneider, S., Treharne, H.: CSP theorems for communicating B machines. Formal Aspects of Computing 17, 390–422 (2005)
55. Shapiro, S.: Specifying and verifying multiagent systems using the cognitive agents specification language (CASL). PhD thesis, Toronto, Ont., Canada, Canada (2005)
56. Shapiro, S., Lespérance, Y., Levesque, H.J.: The cognitive agents specification language and verification environment for multiagent systems. In: AAMAS 2002 [1] , pp. 19–26
57. Shapiro, S., Lespérance, Y., Levesque, H.: The Cognitive Agents Specification Language and Verification Environment. In: Dastani, M., Hindriks, K.V., Meyer, J.-J.C. (eds.) Specification and Verification of Multi-agent Systems, pp. 289–315. Springer, US (2010)
58. Singh, M., Chopra, A.: Correctness Properties for Multiagent Systems. In: Baldoni, M., Bentahar, J., van Riemsdijk, M.B., Lloyd, J. (eds.) DALT 2009. LNCS, vol. 5948, pp. 192–207. Springer, Heidelberg (2010)
59. Telang, P.R., Singh, M.P.: Specifying and Verifying Cross-Organizational Business Models: An Agent-Oriented Approach. TR 12, North Carolina State University (May 2010)
60. Vasconcelos, W., Robertson, D., Sierra, C., Esteva, M., Sabater, J., Wooldridge, M.: Rapid prototyping of large multi-agent systems through logic programming. Annals of Mathematics and Artificial Intelligence 41, 135–169 (2004), doi:10.1023/B:AMAI.0000031194.57352.e7
61. Vasconcelos, W.W., Sabater, J., Sierra, C., Querol, J.: Skeleton-based agent development for electronic institutions. In: AAMAS 2002 [1], pp. 696–703 (2002)
62. Venkatraman, M., Singh, M.P.: Verifying Compliance with Commitment Protocols: Enabling Open Web-Based Multiagent Systems. Autonomous Agents and Multi-Agent Systems 2(3), 217–236 (1999)

63. Viroli, M., Omicini, A.: Process-algebraic approaches for multi-agent systems: an overview. Applicable Algebra in Engineering, Communication and Computing 16, 69–75 (2005), doi:10.1007/s00200-005-0170-3
64. Wooldridge, M.: Agent-Based Software Engineering. IEE Proceedings - Software Engineering 144(1), 26–37 (1997)
65. Wooldridge, M., Fisher, M., Huget, M.-P., Parsons, S.: Model Checking Multi-Agent Systems with MABLE. In: AAMAS 2002 [1], pp. 952–959
66. Wooldridge, M., Jennings, N.R., Kinny, D.: The Gaia Methodology for Agent-Oriented Analysis and Design. Autonomous Agents and Multi-Agent Systems 3(3), 285–312 (2000)

Author Index