A Domain Specific Modeling Language for Multiagent Systems[*]

Christian Hahn
German Research Center for Artificial Intelligence (DFKI)
Stuhlsatzenhausweg 3
66123 Saarbrücken, Germany
Christian.Hahn@dfki.de

ABSTRACT

Software systems are becoming more and more complex with a large number of interacting partners often distributed over a network. A common dilemma faced by software engineers in building complex systems is the lack of clear requirements and domain knowledge needed to come up with a detailed design of the system. Agent technologies are a suitable programming paradigm to cope with the complexity of modern software systems. However, existing agent-based methodologies and tools are developed for experienced programmers and are not suitable for non-agent experts. This paper discusses a domain specific modeling language for multiagent systems that (i) provides a clear syntax and semantics to define agent-based systems in a graphical visualized manner and (ii) can be used to automatically derive code from its design through model transformations.

Categories and Subject Descriptors

I.2.11 [**Distributed Artificial Intelligence**]: Multiagent systems; D.2.11 [**Software Engineering**]: Software Architectures

General Terms

Design, Language

Keywords

Domain Specific Modeling Language, Metamodel, Model-driven Development, Transformations

1. INTRODUCTION

Agent-based computing can be considered as promising approach and powerful technology to develop applications in complex domains by designing and developing applications in terms of autonomous software entities (agents), situated in an environment that can flexibly achieve their goals by interacting with one another in terms of high-level protocols and languages.

Recently, associated with the increasing acceptance of agent-based computing as a novel software engineering paradigm, a lot of research addresses the identification and definition of suitable models and techniques to support the development of complex software systems with respect to agent-based computing. Agent-Oriented Software Engineering (AOSE) is a relatively young field – with its first workshop held in 2000 – that is concerned with how to engineer agent-based software systems.

The current state-of-the-art in developing multiagent systems (MASs) is to design the agent systems basing on an AOSE methodology and take the resulting design artifact as a base to manually code the agent system using agent-oriented programming languages (AOPLs). The gap between design and code may tend to the divergence of design and implementation which makes again the design less useful for further work in maintenance and comprehension of the system [3]. Furthermore, even if existing methodologies have different advantages when applied to particular problems, it seems to be widely accepted that a unique methodology cannot be applied to each problem without some (minor) level of customization.

The framework discussed in this paper presents a platform independent domain specific modeling language for MAS called DSML4MAS that allows modeling agent systems in a platform independent and graphical manner. The models generated conform to the syntax and semantics given by a formal specification language and are thus well-formed. Furthermore, model transformations that base on the principles of model-driven development (MDD) link design and code through model transformations to generate executable code.

The structure of this paper is as follows: Section 2 gives an introduction into language-driven development, followed by Section 3 that discusses a platform independent metamodel for agents that illustrates the core element of our agent-based modeling language. In Section 4, we point out how the semantics of the agent-based modeling language is formulated. Section 5 presents the graphical editor of our modeling language and illustrates how to apply it in a small use case scenario. Section 6 illustrates the core mapping rules to derive executable code from the design artifacts. Section 7 discusses the generated output of these mapping rules based on the presented use case scenario. Related work is presented in Section 8 followed by Section 9 that concludes this paper.

[*]The work published in this paper is partly funded by the European Commission through the ATHENA IP (Advanced Technologies for interoperability of Heterogeneous Enterprise Networks and their Applications Integrated Project) (IST-507849).

Cite as: A Domain Specific Modeling Language for Multiagent Systems, Christian Hahn, *Proc. of 7th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2008)*, Padgham, Parkes, Müller and Parsons (eds.), May, 12-16., 2008, Estoril, Portugal, pp. 233-240.

2. LANGUAGE-DRIVEN DEVELOPMENT

Languages are an essential part of the development of systems that are either used as a high-level modeling language that abstracts from implementation or that are based on a specific implementation technology. Many of these are general-purpose languages like for instance Java or Unified Modeling Language (UML), which provide abstractions that are applicable across a wide variety of domains. In other situations, they will be domain specific languages (DSL) [5] that provide a highly specialized set of concepts to target a small problem domain. In addition to using languages to design and implement systems, languages typically support many different capabilities (e.g. execution and parsing) that are an essential part of the development process.

A strong distinction has traditionally been made between modeling languages and programming languages. One reason for this is that modeling languages have been traditionally viewed as having an informal and abstract semantics whereas programming languages are significantly more concrete due to their need to be executable. However, we will show in this paper that our agent-based modeling language provides key features like a concrete syntax, abstract syntax and semantics that are discussed in more detail in the following.

The *abstract syntax* of a language describes the vocabulary of concepts provided by the language and how they may be combined to create models or programs. It consists of a set of provided concepts, their relationships to other concepts and may also include rules that define whether a model is well-formed. It is important to emphasize that a language's abstract syntax is independent of its concrete syntax and semantics. Abstract syntax deals solely with the form and structure of concepts in a language without any consideration given to their presentation or meaning. This is especially important in the case when additional tool support is provided and the models must be validated before to ensure their correctness. In terms of MDD, the abstract syntax is described by a metamodel that defines how the models should look like. The abstract syntax of DSML4MAS is defined by a platform independent metamodel for MAS called PIM4AGENTS that is discussed in more detail in Section 3.

The *concrete syntax* is the set of notations that facilitates the presentation and construction of the language constructs. The concrete syntax could either be formulated in a textual or visual manner. A textual syntax allows to describe the models in a structured textual form, whereas a visual syntax allows to use a diagrammatical form. For instance, UML uses nodes and edges to represent some underlying model elements. The notations are visualized through figures, e.g. nodes in UML by rectangles and ellipses. The concrete syntax of DSML4MAS is presented in Section 5.

An abstract syntax conveys little information about what the concepts in a language actually mean. Therefore, additional information is needed in order to capture the *semantics* of a language which is important in order to give the language a clear representation and meaning. Otherwise, assumptions may be made about the language that lead to its incorrect use. Even if the developers may have an understanding of the syntax of a language, the semantics are the key to clarify the language's and concept's meanings. In terms of MDD, semantic is often introduced when transforming a platform independent model (PIM) to a specific platform that offers some kind of execution semantics.

In general, a modeling language can be realized in two different ways, either by customization of pre-existing languages through profiles or by creating a new language with a standardized meta-data architecture which defines a set of modeling constructs.

The first approach through customization is achieved by marking up UML concepts with existing stereotypes and tags. The second approach is based on the idea to create a brand new modeling language from scratch. This involves using MDD facilities and standards to create a model of the language which is used to generate a tool for it on an existing platform.

We base our modeling language on the second approach that is constructed in four phases. In the first phase, an adequate level of abstraction must be found. Often, this implies to formally define a language model that specifies the abstract syntax of the language. In terms of our framework, the language model is derived from a formal metamodel. In the second phase, a suitable concrete syntax is defined, e.g. graphical symbols or a textual syntax, which is used by users of the language. As the concrete syntax represents the concepts in the abstract syntax, usually there is a correspondence between concrete and abstract syntax elements. In the third and last phase, a generator translates the modeling language into an executable representation. For this purpose, the elements of the concrete syntax have to be mapped to instances of the abstract syntax, conforming to the formal language model of the agent-based language. From the abstract syntax, code in the target programming language is generated (cf. Section 6).

3. A PLATFORM INDEPENDENT METAMODEL FOR MULTIAGENT SYSTEMS

The challenge in defining a platform independent metamodel that abstracts from existing execution platforms is to decide on the building blocks of MAS. This metamodel becomes the critical element when trying to create a new language as in the agent context, to date, there is no common denominator, as already existing metamodels focus on their own concepts and system structures.

In the following section, platform independent concepts and their attributes are discussed that define the abstract syntax of our agent-based modeling language. However, we do not want to claim that these are the only adequate concepts that should be considered as platform independent, instead we want to show in this paper that this metamodel can be used to define mappings to different AOPLs in order to generate executable code.

The PIM4AGENTS is structured into several aspects each focusing on a specific viewpoint of a MAS. The metamodel bases on Ecore which is the meta-metamodel of the Eclipse Modeling Framework[1] (EMF).

3.1 Multiagent System Aspect

The multiagent system aspect contains the main building blocks of a MAS and thus includes the concepts MultiagentSystem, Agent, Instance, Cooperation, Capability, Interaction, Role, Behavior, and Environment.

[1]http://www.eclipse.org/modeling/emf/

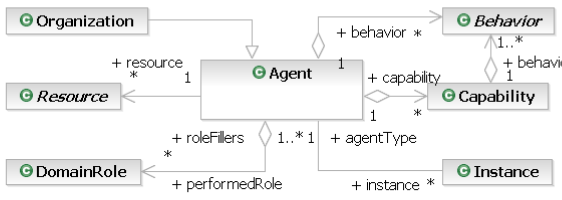

Figure 1: The metamodel reflecting the agent aspect of the Pim4Agents**.**

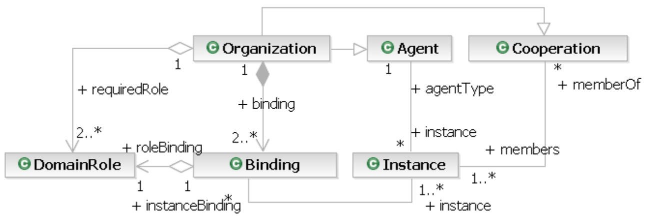

Figure 2: The partial metamodel reflecting the organization aspect of the Pim4Agents**.**

3.2 Agent Aspect

The agent aspect describes single autonomous entities, the capabilities they have to solve tasks and the roles they play. The metamodel of the agent aspect is depicted in Figure 1. It includes the concepts Agent, Capability, and Instance as well as Organization (from the organization aspect), Behavior (from the behavior aspect), Role (from the role aspect), and Resource (from the environment aspect).

This aspect is centered on the concept of Agent, the autonomous entity capable of acting in the system. An Agent has access to a set of Resources which may include information or ontologies from its surrounding Environment. Furthermore, the Agent can perform particular Roles that define in which specific context the Agent is acting and Behaviors defining how particular tasks are achieved. Furthermore, the Agent may have certain Capabilities that group a particular type of Behaviors. To define social structures, an Agent could additionally be member in an Organization that represents the social structure Agents can take part in.

3.3 Organization Aspect

The organization aspect describes how single autonomous entities cooperate within the MAS and how complex organizational structures can be defined. The metamodel of the organization aspect is depicted in Figure 2. It includes the concepts Cooperation, Organization, Protocol (from the interaction aspect), Role (from the role aspect), and Agent (from the agent aspect).

An Organization defines the social structure Agents can take part in. The Organization is a special kind of Cooperation that also has the same characteristics of an Agent. Therefore, the Organization can perform Roles and have Capabilities which can be performed by its members, be it agents or sub-organizations. The multiple inheritance of the Organization, from the Agent and the Cooperation, also allows it to have its own internal Protocol that specifies (i) how the Organization communicates with other Agents be them atomic Agents or complex Organizations and (ii) how organizational members are coordinated.

3.4 Role Aspect

The role aspect covers feasible specializations and how they could be related to each other. The metamodel of the role aspect is depicted in Figure 3. It includes the concepts Role, Actor, and DomainRole as well as Agent, Capability (both from the agent aspect), and Resource (from the environment aspect).

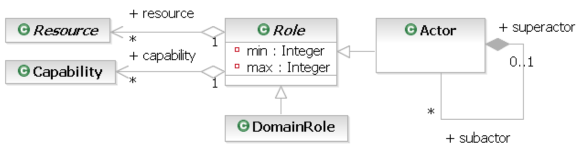

Figure 3: The metamodel reflecting the role aspect of the Pim4Agents**.**

A Role is an abstraction of the social behavior of the Agent in a given social context, usually a Cooperation or Organization. The Role specifies the responsibilities of the Agent in that social context. It refers to (i) a set of Capabilities that define the set of Behaviors it can possess and (ii) a set of Resources the Role has access to. An Actor can be considered as a generic concept as it either binds Instances or DomainRoles. The Actor inherits from the Role and thus can have access to particular Capabilities and Resources that are necessary for exchanging messages. The set of bound entities could be further specialized through the subactor reference that refers again to an Actor. A subactor could be considered as a specialization of the superactor with respect to the Capabilities they provide within an interaction.

A good example why to distinguish between subactors is the Contract Net Protocol [7] (CNP). In the CNP, the *Initiator* sends in the proposal stage either an `accept-proposal` or a `reject-proposal` to the *Participant*. The decision which message is sent depends on the fact if the *Participant* is considered as best bidder with respect to a certain criterion. If this is the case, this *Participant* gets an `accept-proposal`, otherwise a `reject-proposal`. This implicit distinction between best bidder and remaining bidders could be done using Dsml4mas explicit. The Actor *Participant* would contain two subactors, i.e. *BestBidder* and *RemainingBidders* that are filled at run-time where the attributes min and max are specified at design time to illustrate how many fillers are needed at least and at most.

3.5 Interaction Aspect

The interaction aspect describes how the interaction between autonomous entities or organizations takes place. To design flexible and robust agent interactions, message-centric protocols are needed that should enable participating entities to perform appropriate actions. From our point of view, messages should be considered as the least common denominator for interaction but are not sufficient to design robust interaction.

The metamodel of the interaction aspect is depicted in Figure 4. It includes the concepts Protocol, MessageFlow, MessageScope, Message, TimeOut, Operation, and Actor

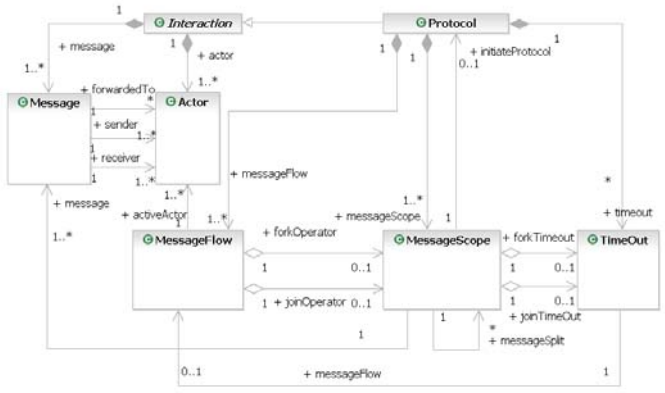

Figure 4: The metamodel reflecting the partial interaction aspect of the Pim4agents.

(from the role aspect). A Protocol refers to (i) at least two Actors that interact within the Protocol, (ii) a set of Messages that are exchanged by the parties concerned, (iii) a set of TimeOuts that define the time constraints for sending and receiving Messages, (iv) a set of MessageFlows that specify how the exchange of messages is to proceed, and (v) a set of MessageScopes that specify the Messages exchanged and if and how the message flow branches. This is done through an Operation that could be of the type None, Sequence, Loop, OR, XOR, and AND.

3.6 Behavior Aspect

The behavior aspect describes how plans are composed by complex control structures and simple atomic tasks like sending a message and how information flows between those constructs. The core metamodel of the behavior aspect is depicted in Figure 5. It includes the concepts Behavior, Plan, Flow, ControlFlow, InformationFlow, Activity, StructuredActivity, and MessageFlow (from the interaction aspect).

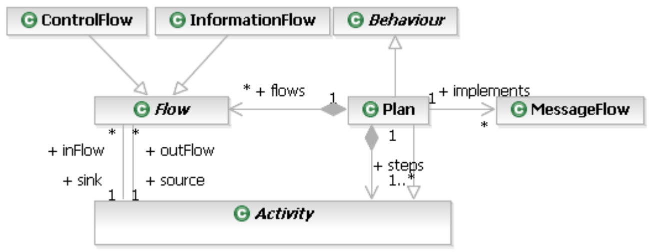

Figure 5: The metamodel reflecting the partial behavior aspect of the Pim4agents.

A Behavior represents the super class connecting the agent aspect with the behavioral aspect, where a Plan can be considered as a specialization of the abstract Behavior to specify an agent's internal processes. A Plan contains a set of Flows and Activities. The Activities are linked to each other via Flows which are either of the type ControlFlow or InformationFlow. Furthermore, a Plan has a reference to a set of MessageFlows that are implemented by the Plan. This means that the internal behavior is defined that is necessary to guarantee the adequate message exchange in accordance to particular Protocol the MessageFlow belongs to.

Beside the concepts that are depicted in Figure 5, several additional concepts have to be mentioned that belong to the behavior aspect and are not part of Fig. 5. These concepts include StructuredActivity and Task that inherit from Activity. The StructuredActivity focuses on complex control structures like Loop, Parallel or Decision whereas Task focuses on atomic activities like sending or receiving a message (i.e. SendMessage, ReceiveMessage). Furthermore, a so called InternalTasks that is a specialization of Task could be used to introduce code specified in Object Constraint Language[2] (OCL).

3.7 Environment Aspect

The environment aspect contains any kind of Resource that is dynamically created, shared, or used by the Agents or Organizations, respectively. A Resource contains a set of Documents that include Classes that refer to Attributes and References. Resources (or Documents that inherit from Resource) are used for exchanging information within Messages or InformationFlows.

4. A FORMAL APPROACH TO REFINE THE SEMANTICS

A metamodel contains only little information about what the concepts in a language actually means and only describes the vocabulary of concepts provided by the language and how they may be combined to create models. Thus, additional information and mechanisms are needed to capture the semantics of a language.

Consequently, we developed a complete formal specification using Object-Z [19]. Object-Z extends Z [22] with object-oriented specification support. The basic construct is the class which encapsulates a state schema with all the operation schemas which may affect its variables. Furthermore, a class includes invariants that specify further restrictions on the variables.

Using Object-Z, we define the abstract syntax and static semantics (ensuring that all concepts are statically well-formed) of the individual concepts in the Pim4agents (e.g. Agent) by formalizing their attributes and invariants. The dynamic semantics is defined by specifying a denotational and operational semantics. The denotational semantics is defined in terms of introducing additional semantic variables and invariants. The operational semantics is defined in terms of class operations and invariants restricting the operation sequences that are specified using the timed trace notation of the timed refinement calculus [20].

5. GRAPHICAL VISUALIZATION AND USE CASE

The concrete syntax of Dsml4mas is specified using the Graphical Modeling Framework[3] (GMF) that provides the fundamental infrastructure and components for developing visual design and modeling surfaces in Eclipse. In principle, GMF allows to define diagrams that base on a mapping between the concrete syntax and the abstract syntax defined

[2]http://www.omg.org/docs/ptc/03-10-14.pdf
[3]http://www.eclipse.org/gmf/

by the Pim4agents metamodel which thus serves as input for the generation of a visual editor. For a complete language description, we extend the generated diagrams by an additional syntax check to guarantee the well-formedness of the generated models. This is done by translating the Object-Z specification into OCL rules (as explained in [18]) that are used to validate and check the created models at design-time to ensure that the models can be mapped to the AOPLs. The models that were created using the concrete syntax conform to the Pim4agents and its semantics and could thus be used as input for the model transformation to generate code.

In the following, we discuss the concrete syntax in a small example that covers a conference management system (CMS). We assume that the reader is familiar with the process of submitting a paper to an international conference (e.g. AAMAS). This process starts with a call for papers (CFP) distributed by the program committee (PC). When receiving the CFP, authors decide whether to submit a paper. After the deadline has passed, the PC distributes all received papers among the PC members that are in charge of providing a review for their assigned papers that is sent back to the PC. Considering all reviews, the PC decides on the accepted papers and sends a message to the corresponding authors to inform them about acceptance or rejection. To keep this example simple, we mainly concentrate on the submission phase in the following.

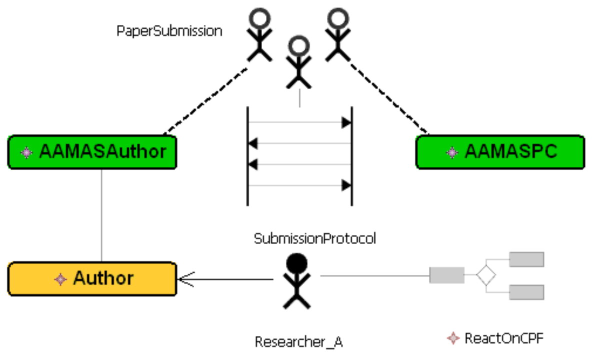

Figure 6: The agent model of the CMS use case exported from the agent diagram.

Figure 6 depicts the agent diagram of the CMS case. The model consists of an Agent called *Researcher_A* that performs the DomainRole of an *Author* and acts in accordance to a Plan called *ReactOnCPF*. This Plan defines in which manner the *CallForPapers* is handled, according to which criterion the Agent decides whether it submits a paper or not and finally, how it replies to the initially sent *CallForPapers*. The message exchange is illustrated in the interaction diagram (see Figure 7). Due to space restriction, the behavior aspect of the CMS use case is not discussed in more detail in this paper. The DomainRole *Author* is bound to an Actor (this is allowed as the Actor inherits from Role) called *AAMASAuthor* that is required – in addition to the Actor *AAMASPC* – by the Organization *PaperSubmission*. The participants of this Organization are coordinated in accordance to a Protocol *SubmissionProtocol* that defines the message exchange between both parties.

The SubmissionProtocol is depicted in Figure 7. It con-

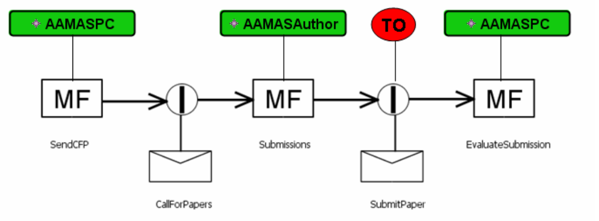

Figure 7: An interaction model of the CMS use case exported from the interaction diagram.

sists of two Messages called *CallForPapers* and *SubmitPaper*. Both Messages are attached to a MessageScope that uses the None Operator expressing that the message flow is not split. The MessageFlow *SendCFP* in which the *AAMASPS* is active is the source for sending the *CallForPapers*, whereas the MessageFlow *Submission* in which the *AAMASAuthor* is active receives the *CallForPapers* and sends the Message *SubmitPaper* if the *Researcher_A* has decided to submit a paper. This decision is implemented as a StructuredActivity within the *ReactOnCFP* Plan. The TimeOut *TO* represents the deadline for submitting a paper to the *AAMASPC*.

6. CODE GENERATORS

Model transformations are one of the key mechanisms within MDD. Basing on code generation templates, the model is transformed to executable code that may optionally be merged with manually written code. One or more model-to-model as well as a model-to-text transformation steps could be necessary for code generation. The implementation of model-to-model transformations is done using the Atlas Transformation Language[4] (ATL) that again bases on the Ecore meta-metamodel.

In the context of our approach, we developed model transformations to two target languages (i.e. Jack [13] and JADE [1]) and currently investigate a model transformation between the Pim4agents and Jadex [16]. The main motivation for choosing the mentioned AOPLs is their different view on agent systems. Whereas Jack and Jadex base on principles of the Belief-Desire-Intention (BDI, cf. [17]) architecture, JADE focuses on the compliance with the FIPA[5] specifications for interoperable intelligent MASs and thus concentrates on interaction aspects. A mapping from the Pim4agents's concepts to the concepts of the different execution platform demonstrates that the concepts of the Pim4agents can be considered as platform independent.

Due to space restrictions, we focus in this paper on the Pim4agents to Jack transformation. For detailed information regarding the Pim4agents to JADE transformation, we refer to [9].

In the remainder of this section, several mapping rules (depicted in Figure 8) are discussed that transfer the Pim4agents metamodel to the metamodel of Jack (JackMM). For detailed information regarding JackMM and the model transformation we refer to [9, 8].

Mapping Rule 1: *Organization* → *Team*
The source and target concepts of this mapping rule nicely

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

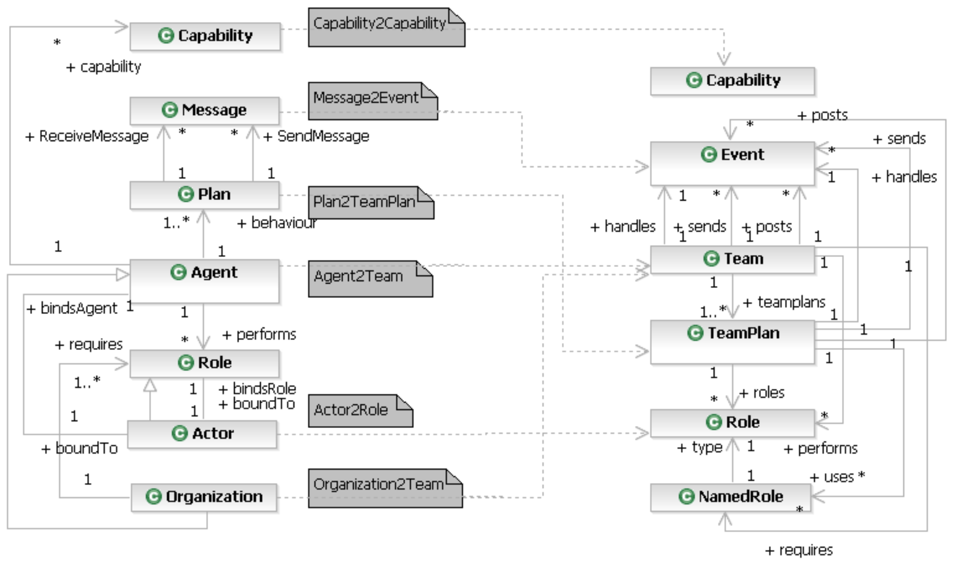

Figure 8: The basic mapping rules of the Pim4Agents (left hand side) to the metamodel of Jack (JackMM) (right hand side) transformation. Please note that both metamodels only present abstractions of the originals. For instance, a Plan in the Pim4Agents does not directly refer to Messages, instead it contains the concepts SendMessage and ReceiveMessage that directly refer to Messages.

correspond to each other as both (i) make use of an internal behavior that specifies how their members are coordinated and (ii) require and perform Roles. The Behavior as well as the Roles could thus be easily mapped from the Pim4Agents to Jack. The only difference between both metamodels is the manner in which interactions are defined, i.e. the interaction in the Pim4Agents is defined by a Protocol whereas JackMM defines the interaction between entities in an event-driven manner without explicitly specifying a protocol.

Mapping Rule 2: *Agent → Team*
At first glance the concept Agent of JackMM seems to be the best match, but since an Agent in the Pim4Agents references Roles, it is recommended to assign an Agent (from Pim4Agents) to an atomic Team – which does not require any NamedRole – in JackMM. The Behaviour or rather the concrete Plan used by the Agent is mapped to a set of TeamPlans the Team makes us of. The Messages of the Protocol the Agent participates are mapped to Events that are either handled or sent/posted by the Team. Furthermore, the Team performs the Roles that are performed by the Agent in the Pim4Agents.

Mapping Rule 3: *Plan → TeamPlan*
A TeamPlan uses a set of NamedRoles that are extracted from the Roles the corresponding Organization requires in the Pim4Agents. In fact, only an Organization (and Cooperation from which the Organization inherits) requires Roles, i.e. the set of NamedRoles an atomic Team requires is empty. Detailed information on how the Plan's elements are mapped is discussed in [8].

Mapping Rule 4: *Message → Event*
Each Message that is either part of a *Protocol* or is referred by an atomic Task (i.e. SendMessage or ReceiveMessage) in a *Plan* is mapped to an Event (i.e. MessageEvent) in JackMM.

Mapping Rule 5: *Capability → Capability*
The Behavior that is used by the Capability in the Pim4Agents is mapped to the Capability's Plan in JackMM. The Messages that are sent and received within the particular Behavior are mapped to Events that are sent and handled by the Capability in JackMM.

Both kinds of Capabilities nicely correspond to each other. However, in JackMM only the concepts Agent and Team refer to Capabilities, but not Roles. To compensate this, we introduce additional Capabilities for those Teams that perform the particular Roles referring to Capabilities in the Pim4Agents.

Mapping Rule 6: *Actor → Role*
The concept Actor of the Pim4Agents is transformed to Jack-related Roles. The Instances or Roles that are bound to the particular Actor are used as role fillers a Team can make use of in Jack.

The model-to-model transformation generates an output model in accordance to JackMM which serves as input model for the model-to-text transformation that generates Jack Gcode (XML-like documents that can be imported into the Jack IDE). This transformation is implemented using MOFScript[6] that bases – like ATL – on Ecore. In MOFScript, serialization rules (i.e., templates) are created following the structure of JackMM, i.e the information regarding the concept itself as well as the references to other concepts are extracted from the JackMM model and assigned to the template's attributes. Consequently, a template is created for the concepts Event, Role, NamedRole, Agent, Plan, Team and TeamPlan. For each instance of these concepts in the

[6]http://www.eclipse.org/gmt/mofscript

JackMM model, a new file is generated that could be imported into the Jack development IDE and compiled to generate executable Jack code.

7. GENERATED JACKMM MODELS

In the previous section, we illustrated the basic mapping rules used to transform Pim4agents models to JackMM models and described how to generate Jack code based on these JackMM models. For the purpose of demonstration, we relate the model transformation to the Pim4agents models that were discussed in Section 5 and explain how the generated Jack models look like.

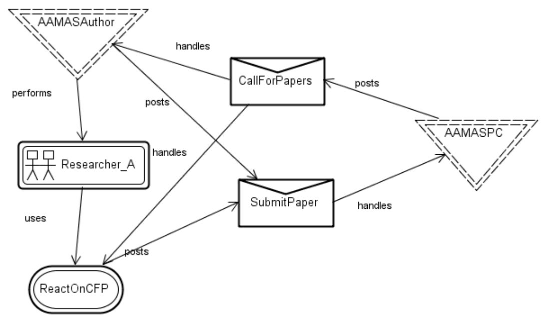

Figure 9: The generated Jack model.

Figure 9 depicts the generated Jack model. The Team *Researcher_A* is generated by applying Mapping Rule 2 on the Agent *Researcher_A*. The Team *Researcher_A* uses a TeamPlan called *ReactOnCPF* that is generated by applying Mapping Rule 3 on the Plan *ReactOnCPF*. Furthermore, Mapping Rule 6 instantiates two Roles (i.e. *AAMASAuthor* and *AAMASPC*), where the *AAMASAuthor* is performed by the Team *Researcher_A*. These Roles handle and post Events called *CallForPapers* and *SubmitPaper*, that are generated by applying Mapping Rule 4.

8. RELATED WORK

Several agent-oriented methodologies have been already proposed [23, 15]. Obviously, a comprehensive study and comparison of these is out of scope of this paper. Nevertheless, a short overview on already well-established methodologies in the agent community is given in the remainder of this section.

Tropos [4] is an agent-oriented methodology based on the concepts of Actor and Goal. The Tropos methodology consists of five phases including early and late requirements. The core concepts of the Tropos metamodel are Actor and its specializations Position, Agent and Role, where a Position covers at least one Role and an Agent may play Roles and may occupy Positions. Furthermore, an Agent may want Goals that are either HardGoals or SoftGoals. Tropos is supported by a graphical visualization tool called *Tool for Agent Oriented Visual Modeling (TAOM)* [14] that allows to generate models in accordance to the Tropos metamodel on the different phases. Furthermore, by adopting principles of MDD, TAOM is able to perform a model-to-model transformation on the generated models to produce JADE code. However, Tropos does not provide support for transforming design into executable agent code as already asserted by [10].

Prometheus [12] defines a detailed process for analysis, design, and implementation. The core concepts of the Prometheus metamodel are (i) Goal, (ii) Role that achieve Goals, has access to Data and has to handle Percepts, and (iii) Agent that may have access to a set of Capabilities, owns a set of Plans and plays a set of Roles. Furthermore, an Agent may be either participant or initiator of a Protocol/Interaction which includes Messages. Like Tropos, Prometheus is supported by the java-based graphical editor *Prometheus Design Tool (PDT)* [21] and the eclipse-based graphical editor *Agent Development Tool Plug-in for Eclipse (ADPT)* [11] that both allow to automatically generate code in BDI based AOPLs like Jack and Jadex. However, none of them generate design artifacts that capture sufficient details to apply automated generators producing executable code.

The Unified MAS Metamodel proposal [2] was the first attempt towards the development of a unified metamodel. This metamodel was developed by merging the metamodels of ADELFE [15], Gaia [23] and PASSI [6] and thus combining the strengths of each metamodel. The unified metamodel covers aspects such as (i) cooperative behavior as described by the ADELFE metamodel, (ii) organizational behavior as specified by the Gaia metamodel, and (iii) FIPA-compliant communication structures as defined by the PASSI metamodel. Even if this metamodel defines the most important building blocks of agent systems, it is not really clear if executable code can be generated as neither the internal behavior of an agent nor the external behavior – i.e. the agent interaction – are specified in an adequate manner.

9. CONCLUSION

This paper presented a platform independent modeling language for agents in the context of MDD with the objective to close the gap between design and code. In this context, we have identified the following advantages of our approach:

The Dsml4mas covers the core building blocks of an agent system. The identified building blocks enable a mapping to different AOPLs like Jack and JADE which is – to the best of our knowledge – not supported by any other agent-oriented modeling language.

A formal description of the Pim4agents is given using Object-Z to (i) further refine its abstract syntax and (ii) formalize the denotational and operational semantics.

The syntax and semantics defined are used as a base to develop a graphical editor that finally formulates the concrete syntax. Syntax and semantics are expressed with OCL to guarantee that the developed models are well-formed. This is of special importance when applying the model transformation to the specific AOPLs.

MDD has the potential to addresses interoperability issues necessary to link design and code. Exemplarily, this is shown by the model transformation between Dsml4mas and Jack.

In [24, 8], we showed how to integrate service-oriented architectures (SOAs) into JackMM in a model-driven scenario. The agent-based modeling language could easily be integrated into this approach to execute SOAs. It is not clear whether Prometheus or Tropos would allow this kind of integration as the concepts used for instance in the early requirement phase of Tropos only cover Actor and Goal which might not be sufficient for deploying (semantic) web services.

However, the Pim4agents and thus the Dsml4mas should not be considered as completed, as mentioned before, it could easily be extended and refined.

Concepts like beliefs and commitments could be added in the next versions. Up to now, our MDD approach only considers a top-down approach from platform independent to platform specific models. Thus, any change of the executable code cannot be propagated to the Dsml4mas. This is an interesting approach that will be explored in the future work.

10. REFERENCES

[1] F. Bellifemine, F. Bergenti, G. Caire, and A. Poggi. *JADE - a java agent development framework.*, volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*, chapter 5, pages 125–147. Springer, Berlin et al., 2005.

[2] C. Bernon, M. Cossentino, M.-P. Gleizes, P. Turci, and F. Zambonelli. A study of some multi-agent meta-models. In J. Odell, P. Giorgini, and J. Müller, editors, *Agent-Oriented Software Engineering V: 5th International Workshop, AOSE 2004. Revised Selected Papers*, Lecture Notes in Computer Science 3382, pages 62–77. Springer-Verlag, 2005.

[3] R. H. Bordini, M. Dastani, and M. Winikoff. Current issues in multi-agent systems development (invited paper). In *Post-proceedings of the Seventh Annual International Workshop on Engineering Societies in the Agents World*, volume 4457 of *Lecture Notes in Artificial Intelligence*, pages 38–61, 2007.

[4] P. Bresciani, A. Perini, P. Giorgini, F. Giunchiglia, and J. Mylopoulos. TROPOS: An Agent-Oriented Software Development Methodology. *Journal of Autonomous Agents and Multiagent Systems*, 8(3):203–236, 2004.

[5] S. Cook, G. Jones, S. Kent, and A. C. Wills. *Domain-Specific Development with Visual Studio DSL Tools*. Addison-Wesley Professional; 1 edition, 2007.

[6] M. Cossentino. From requirements to code with the PASSI methodology. In B. Henderson-Sellers and P. Giorgini, editors, *Agent-Oriented Methodologies*, Hershey, PA, USA, 2005. Idea Group Inc.

[7] R. Davis and R. G. Smith. Negotiation as a metaphor for distributed problem solving. *Artificial Intelligence*, 20:63–109, 1983.

[8] K. Fischer, C. Hahn, and C. Madrigal-Mora. Agent-oriented software engineering: a model-driven approach. *Int. J. Agent-Oriented Software Engineering*, 1(3/4):334–369, 2007.

[9] C. Hahn, C. Madrigal-Mora, and K. Fischer. A platform-independent model for agents. Technical Report RR-07-01A, German Research Center for Artificial Intelligence (DFKI GmbH), Stuhlsatzenhausweg 3, D-66123 Saarbrücken, 2002.

[10] G. Jayatilleke, J. Thangarajah, L. Padgham, and M. Winikoff. Component agent framework for domain-experts (CAFnE) toolkit. In *Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pages 1465–1466, New York, 2006. ACM Press.

[11] L. Padgham, J. Thangarajah, and M. Winikoff. Tool support for agent development using the prometheus methodology. In *Proceedings of the Fifth International Conference on Quality Software*, pages 383–388, Washington, 2005. IEEE Computer Society.

[12] L. Padgham and M. Winikoff. Prometheus: A Methodology for Developing Intelligent Agents. In F. Giunchiglia, J. Odell, and G. Weiß, editors, *Agent-Oriented Software Engineering (AOSE-2002)*, volume 2585 of *Lecture Notes in Computer Science*, pages 174–185, Berlin et al., 2002. Springer.

[13] M. Papasimeon and C. Heinze. Extending the UML for designing JACK agents. In *Proceedings of the Australian Software Engineering Conference (ASWEC 01)*, 2001.

[14] L. Penserini, A. Perini, A. Susi, and J. Mylopoulos. From stakeholder intentions to software agent implementations. In *Proceedings of the 18th Conference on Advanced Information Systems Engineering*, volume 4001 of *LNCS*. Springer Verlag, 2006.

[15] G. Picard and M.-P. Gleizes. *Methodologies and Software Engineering for Agent Systems, The Agent-Oriented Software Engineering Handbook*, chapter 8, The ADELFE Methodology. Kluwer Academic Publishers, 2004.

[16] A. Pokahr, L. Braubach, and W. Lamersdorf. *Jadex: A BDI Reasoning Engine*, volume 15 of *Multiagent Systems, Artificial Societies, and Simulated Organizations*, chapter 6, pages 149–174. Springer, Berlin et al., 2005.

[17] A. S. Rao and M. P. Georgeff. Modeling agents within a BDI-architecture. In R. Fikes and E. Sandewall, editors, *Proceedings of the 2rd International Conference on Principles of Knowledge Representation and Reasoning (KR'91)*, pages 473–484, Cambridge, Mass., 1991. Morgan Kaufmann.

[18] D. Roe, K. Broda, and A. Russo. Mapping uml models incorporating OCL constraints into Object-Z. Technical Report 2003/9, Imperial College, 180 Queen's Gate, London, 2002.

[19] G. Smith. *The Object-Z Specification Language. Advances in Formal Methods*. Kluwer Academic Publishers, 2000.

[20] G. Smith and I. J. Hayes. Structuring real-time Object-Z specifications. In *Proceedings of the Second International Conference on Integrated Formal Methods*, volume 1945 of *Lecture Notes In Computer Science*, pages 97–115, London, 2000. Springer.

[21] J. Thangarajah, L. Padgham, and M. Winikoff. Prometheus design tool. In *Proceedings of the fourth international joint conference on Autonomous agents and multiagent systems*, pages 127–128, New York, 2005. ACM Press.

[22] J. Woodcock and J. Davies. *Using Z: Specification, Refinement, and Proof*. Prentice-Hall, 1996.

[23] F. Zambonelli, N. Jennings, and M. Wooldridge. Developing multiagent systems: the Gaia methodology. *ACM Transactions on Software Engineering and Methodology*, 12(3):417–470, 2003.

[24] I. Zinnikus, C. Hahn, M. Klein, and K. Fischer. An agent-based, model-driven approach for enabling interoperability in the area of multi-brand vehicle configuration. In B. J. Krämer, K.-J. Lin, and P. Narasimhan, editors, *ICSOC*, volume 4749 of *Lecture Notes in Computer Science*, pages 330–341. Springer, 2007.