Electronic Communications of the EASST
Volume 3 (2006)

Proceedings of the Third Workshop on Software Evolution through Transformations: Embracing the Change (SeTra 2006)

A MDE-Based Approach for Developing Multi-Agent Systems

Viviane Torres da Silva, Beatriz de Maria, Carlos J. P. de Lucena

14 Pages

Guest Editors: Jean-Marie Favre, Reiko Heckel, Tom Mens
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

A MDE-Based Approach for Developing Multi-Agent Systems

Viviane Torres da Silva Silva*, Beatriz de Maria, Carlos J. P. de Lucena****

*Departamento de Sistemas Informáticos y Programación, UCM, Spain
** Departamento de Informática, PUC-Rio, Brazil
viviane@fdi.ucm.es, {biamar, lucena}@inf.puc-rio.br

Abstract: This paper focuses on the development of multi-agent systems based on a model driven engineering approach. Our goal is to cope with the traceability between design and implementation models and with the always changing characteristics of such systems.

Keywords: model driven architecture, multi-agent systems, transformation, evolution

1 Introduction

The development of multi-agent systems (MAS) has rapidly increased in the last few years. Modeling languages [13][17][22], platforms [16][18], methodologies [2][4][15] and some other MAS modeling and implementing techniques have been proposed with the purpose of helping the developers in building such systems. Although several approaches are concerned with modeling and implementing MAS, only few accomplish the tracking between design models and implementation code.

While dealing with MAS, the refinement of design models into implementation code becomes especially difficult since it is necessary to deal with different paradigms during the system development. The agent-oriented paradigm is used while modeling the systems but, frequently, those systems are implemented by using the object-oriented paradigm (OO). Since agents and objects have different properties and characteristics (for instance, agents are autonomous and goal-oriented entities that execute plans in order to achieve their goals and, different from objects, do not need external stimulus to execute and can ignore requests), the transformation from agent-oriented design models into OO code is not simple. To try to assist the implementation of MAS, several OO platforms, architectures and frameworks such as [18][16] have been proposed. Although such approaches satisfactory provide satisfactory support for the implementation of the MAS, they fail to provide the tracing between the design models and the implementations. The numerous MAS modeling language, methodologies and platforms deal with different agent properties and characteristics what directly impact in the traceability.

Another important concern that mostly affects the development of MAS is the always changing characteristic of MAS applications and techniques. Since a number of fundamental questions about the nature and the use of the agent-oriented approach are still being answered, important techniques' features and also applications' requirements are still evolving.

This paper focuses on helping the MAS developers to cope with (i) the mapping between design models and implementation models and with (ii) the always changing characteristics of the MAS applications and techniques. In order to achieve these two goals, we use a model driven engineering (MDE) approach [11] for developing MAS. Being our proposal a MDE approach we specify (i) the modeling languages being used to describe the

source and target models, (ii) both models, (iii) the transformations, and (iv) some guidelines to cope with models evolution [11].

The traceability between MAS design models and OO implementation models will be illustrated by the use of the multi-agent system modeling language called MAS-ML [17][19] (the source modeling language), the agent society framework called ASF [18] and the UML modeling language [14] (the target modeling language). The MAS-ML design models (the source models) will be transformed into UML implementation models (the target models) by instantiating the ASF framework according to the application characteristics.

Our second goal is accomplished by demonstrating that the transformation rules can be adapted to the evolution of both applications and techniques being used in the transformation. Such adaptation is facilitated due to the low coupling between design and implementation models. Design models are independent of the platform / framework being used to implement the system, i.e., design models are not concerned with any characteristic of the implementation technique. In addition, the framework is also defined completely independent of the modeling language being used in the design models.

The paper is organized as follows. In Section 2 we present the MAS techniques being used in the paper: MAS-ML and ASF. Section 3 introduces the transformation process and Section 4 provides some guidelines to embrace the evolution of applications and techniques. Section 5 describes some related work and Section 6 draws the conclusions and discusses future work.

2 Multi-Agent System Techniques

The development process of complex and large-scale systems, such as MAS, involves the construction of different models based on a variety of requirements. The transformation of a system specification into models and of these models into code is usually accomplished in a non-organized way that is not easily adaptable to technology changes. In fact, it is possible to find different modeling languages, methodologies and platforms for modeling and implementing MAS but it is hard to find in the literature approaches that trace the design models into code. Therefore, we propose in this paper a top-down MDE approach that traces MAS-ML design models into ASF object-oriented code.

The MAS-ML Modeling Language

MAS-ML is a platform independent modeling language that extends UML incorporating agent-oriented abstractions (such as agents, organizations, environments and roles), their properties (such as goals, plans, actions, beliefs and protocols) and relationships (such as play, inhabit and ownership). Although MAS-ML uses agent and object-oriented abstractions, MAS-ML does not restrict the implementation of its models to a specific implementation platform.

Figure 1 illustrates an important part of the MAS-ML metamodel that presents the agent and object-oriented abstractions defined in the metamodel and the possible relationships among them. By using MAS-ML it is possible to model, for instance, the roles that agents can play (by using a static diagram defined in MAS-ML called organization diagram) and agents achieving their goals while executing their plans (by using the extended UML sequence diagram defined in MAS-ML).

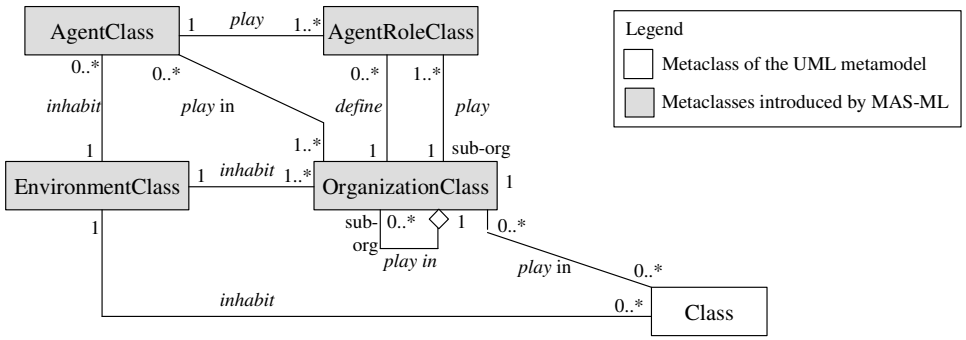


Figure 1. Part of the MAS-ML metamodel

The ASF Framework

The goal of ASF is to help designers implementing MAS by using the object-oriented paradigm. The framework defines a set of OO models where each model represents a MAS entity type. The set of OO classes and relationships defined in each module makes possible the implementation of the structural aspects of MAS entities (its properties and relationships with other entities) and also the dynamic ones (their behavior).

Figure 2 shows a UML class diagram modeling all ASF classes grouped by modules. The modules that are used to instantiate agents (delimited by a continuous rectangle), organizations (defined by the dotted enlace) and agent roles (marked by the hatched rectangle) are complex modules since they group several classes to represent all the properties of these entity types. The module that corresponds to environments (hatched circle) is simple represented by one class because the properties of this entity can be directly represented as attributes and methods.

In order to use ASF to implement a MAS application, it is necesary to instantiate the framework by extending the defined modules. The extentions should be made according to the entities characteristics defined in the application being implemented. For instance, to implement application agents by using ASF it is necessary (i) to create an OO class extending the *Agent* abstract class defined in the ASF agent module to be used to instantiate the agents, (ii) to create OO classes by extending the *Plan* and *Action* classes to implement the plans and actions of the agents, (iii) to implementing the constructor method of the new agent class to create the (instances of the) beliefs, goals and plans and also (iv) to relate the agent instances to the roles that they will play, the organizations where they will play such roles and also to the environments that they will inhabit. Similar steps could be followed in order to implement the organizations, roles and environments defined in the MAS application.

3 The Transformation Process

In this section we describe the transformation process used to refine MAS-ML design models into UML implementation models that instantiate ASF. The transformations were defined by using the Atlas Transformation Language (ATL) [9]. An ATL transformation program is composed of rules (described in ATL) that define how source model elements are matched and navigated to create and initialize the elements of the target models. Besides the rules, an ATL program receives as input (i) the metamodel of the source model (the MAS-ML metamodel), (ii) the source model itself (a MAS-ML model) and (iii) the metamodel of the target model (the UML metamodel). The program checks the source model according to its metamodel and,

by using the transformation rules (MAS-ML2ASF rules), transforms the source model into the output target model (UML model) that is compliant with the target metamodel. In MDE [11], transformation processes based on the translations between metamodels are called *language translations*. Figure 3 illustrates the inputs and the output of the ATL program that transforms MAS-ML models into UML models by using the MAS-ML2ASF rules.

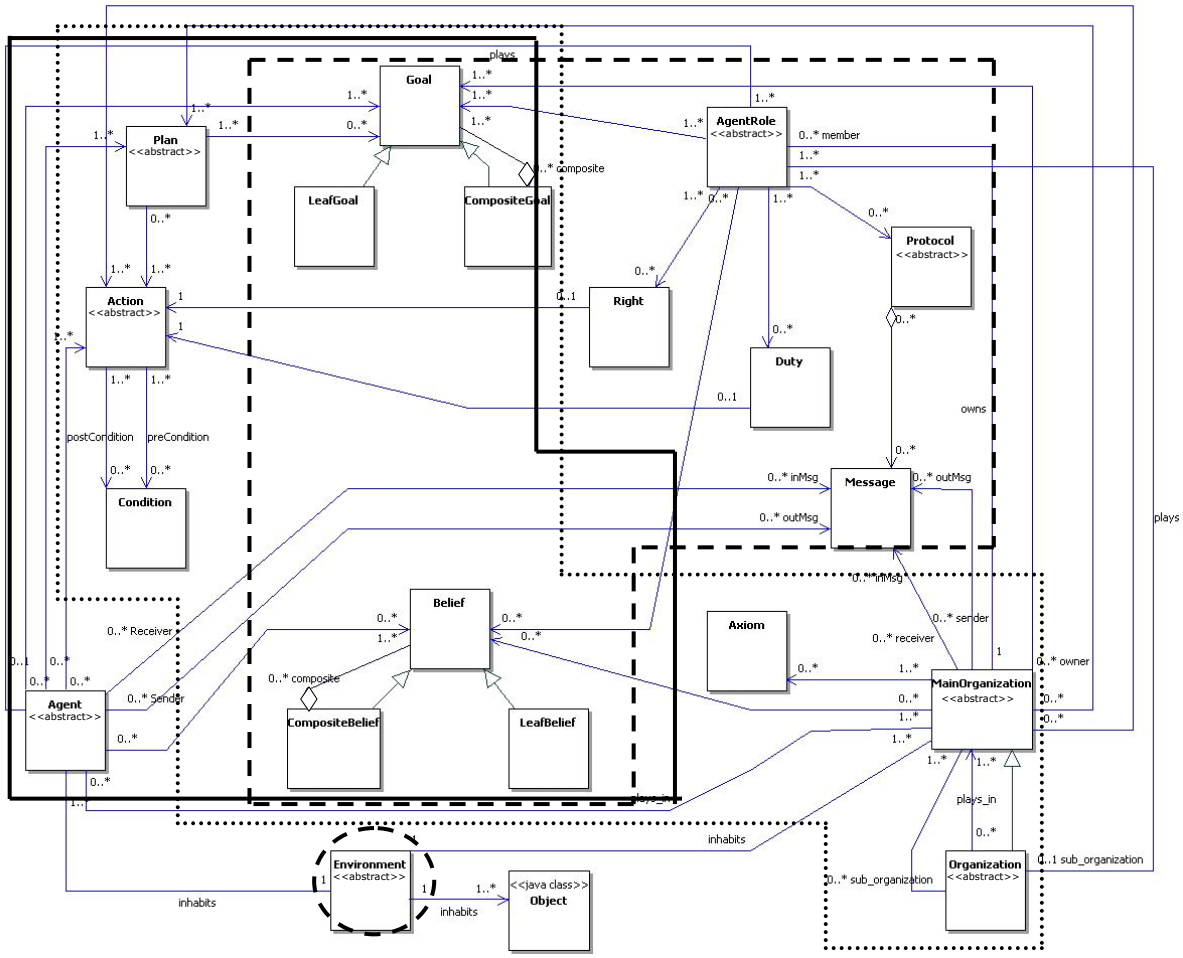


Figure 2. The ASF classes and relationships

The set of rules used by the ATL program to transform a MAS-ML model into a UML model instantiates the ASF framework according to the application features modeled in MAS-ML. The set of MAS-ML2ASF rules is composed of (i) one general rule that generates all the classes defined in ASF (and illustrated in Figure 2); (ii) five rules to transform the five entity types found in MAS-ML models (and defined in the MAS-ML metamodel depicted in Figure 1); and (iii) three other rules to generate the concrete plans, actions and protocols that extends the abstract classes *Plan*, *Action* and *Protocol* defined in ASF (also illustrated in Figure 2).

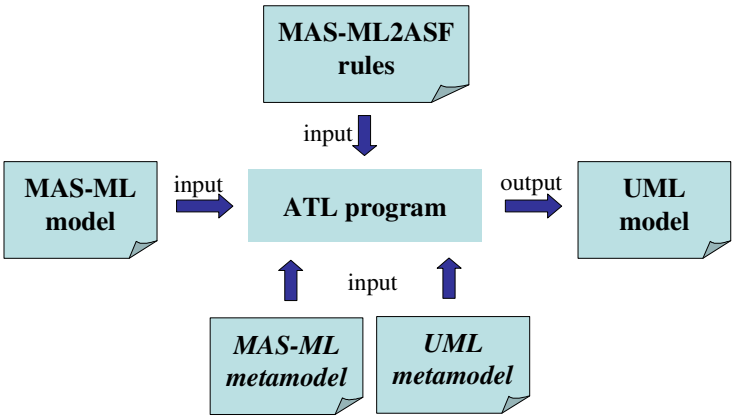


Figure 3. The ATL transformation process

The Transformation Rules

Due to page limitation, we concentrate on demonstrating the transformation of agents modeled using MAS-ML into OO classes by instantiating ASF. The following sections describe the MASML2ASF rules used by the ATL program that contemplates such transformation.

The creation of the concrete agent class

For each distinguished agent class modeled in a MAS-ML structural diagram, one object-oriented agent class that extends the abstract ASF class called *Agent* must be created. Therefore, we have implemented a rule that is executed for each agent defined in MAS-ML. Such rule, illustrated in the box below, is responsible for creating the agent class and for implementing the constructor method (detailed in Section 0).

The new classes are created with the same name of the agent classes modeled in MAS-ML models. Associated with each agent class, we define a generalization relationship in order to model the new agent class extending the abstract *Agent* class defined in ASF. Due to such generalization, the new agent class inherits all the attributes (and relationships) defined in the abstract class.

The set of features of the new class is composed of (i) the constructor method, (ii) methods for instantiating goals, beliefs, plans and actions, and (iii) methods for creating the associations between agent instances and the role instances to be played, and the associations between the agent instances and the environment instances.

```
rule Agent {
from
        c : masml!AgentClass
to
----------------------------
-- Creating the agent class
outAgent : uml!Class (
        name <- c.name, --the name of the new class is equal to the name of the agent
        isAbstract <- false, --the new class is not an abstract one
        generalization <- genAbstractAgent,   --creating the generalization relationship
        feature <- Set{constMethod, createBeliefs, createGoals, createPlans,
createActions, createInhab, createPlays}), --creating the set of structural and
behavioral features
-------------------------------------------------------------------------
-- Creating the generalization relationship between Agent and UserAgent
genAbstractAgent : uml!Generalization (
        parent <- c.superClass,
        child <- outAgent
```

```
),...}
```

The implementation of the constructor method

The constructor method of agent classes invokes other methods that are used to instantiate the agent properties (goals, beliefs, plans and actions) and to set the relationships between the agents instances, their roles, and environments. Therefore, the body of this method is very simple.

```
----------------------
-- Constructor Method
constMethod : uml!Method (
        specification <- operationAgent,
        body <- 'createBeliefs(); createGoals(); createPlans(); createActions();
createPlays();
                createEnv(); createPlays();'
),
operationAgent : uml!Operation (
        isAbstract <- false,
        specification <- 'UserAgent()'
),
```

The instantiation of the agent properties are exemplified by describing the instantiation of beliefs and plans. The beliefs and goals of an agent are created by instantiating the ASF classes *Belief* and *Goal*, respectively. Therefore, the body of the method responsible for generating the beliefs creates one new belief instance from the *Belief* class for each belief defined by the agent classes of the MAS-ML model being transformed.

The plans and actions of an agent are created by instantiating the classes that correspond to the plans and actions themselves. Those classes are extensions of the *Plan* and *Action* classes as demonstrated in Section 0.

```
-------------------------------------
-- Instantiating the agent beliefs
createBeliefs : uml!Method (
        specification <- operationCreateBeliefs,
        body <- c.beliefs -> iterate (e; body : String = '' | body+'
                Belief newBelief = null;
                beliefs = new Vector();
                newBelief = new Belief('+e.type+','+e.value+'); -- instantiating a
belief
                beliefs.add(newBelief);')
),
operationCreateBeliefs : uml!Operation (
        isAbstract <- false,
        specification <- 'void createBeliefs()'
),
---------------------------------
-- Instantiating the agent plans
createPlans : uml!Method (
        specification <- operationCreatePlans,
        body <- c.plans -> iterate (e; body : String = '' | body+'
                Plan newPlan = null;
                plans = new Vector();
                newPlan = new '+e.name+'(); -- instantiating a plan
                plans.add(newPlan);')
),
operationCreatePlans : uml!Operation (
        isAbstract <- false,
        specification <- 'void createPlans()'
),
```

After instantiating the agent properties, it is necessary to relate the agent instances with the roles to be played, the organization where the roles will be played and the environments that they will inhabit. In order to do so, two methods were created. Both methods are called by the agent constructor method every time an agent instance is created.

```
---------------
```

```
-- Environment
createInhab : uml!Method (
        specification <- operationInhab,
        body <- c.inhRel -> iterate (e; body : String = '' | body+'
        this.environment = new '+e.env.name+'();')   --environment
),
operationInhab : uml!Operation (
        isAbstract <- false,
        specification <- 'void createEnv()'
),
----------------------
-- Roles being played
createPlays : uml!Method (
        specification <- operationPlays,
        body <- c.playRelAg -> iterate (e; body : String = '' | body+'
                //roles being played
                AgentRole newRole = null;
                rolesBeingPlayed = new Vector();
                newRole = new '+e.role.name+'();-- instantiating the role
                newRole.setAgent(this);
               newRole.setOrganization(\''+e.org.name+'\');-- associating with the
organization
                rolesBeingPlayed.add(newRole);
                // organizations where is playng roles
                MainOrganization newOrg = null;
                organizations = new Vector();
                newOrg = new '+e.org.name+'();
                newOrg.setAgentRole(\''+e.role.name+'\');
                organizations.add(newOrg);')
),
operationPlays : uml!Operation (
        isAbstract <- false,
        specification <- 'void createPlays()'
)
```

The creation of the plans and actions classes

Each agent defines its set of plans and their correspondent actions. In the MAS-ML structural diagrams, the plans and actions are named and associated with the agent. The execution of plans and actions are therefore modeled in MAS-ML dynamic diagrams.

In order to create the correspondent plans and actions by using ASF, it is necessary to create the classes that will represent these agent properties by extending the abstract ASF classes *Plan* and *Action*. Those classes receive the name of the plans and actions defined in the MAS-ML structural diagrams and also the implementation described in the MAS-ML dynamic diagrams. Note that plans are related to the goals that they achieve and to the actions that they execute. Therefore, the constructor method of a plan executes methods to relate the plan instance being created to its goals and actions.

```
rule Plan {
from
        c : masml!Plan
to
outPlan : uml!Class (
        name <- c.name,
        isAbstract <- false,
        feature <- Set{constMethod, createActions, createGoals}
),
constMethod : uml!Method (
        specification <- operationPlan,
        body <- 'createActions(); createGoals()'
),
operationPlan : uml!Operation (
        isAbstract <- false,
        specification <- c.name+'()'
),
---------------------------------
```

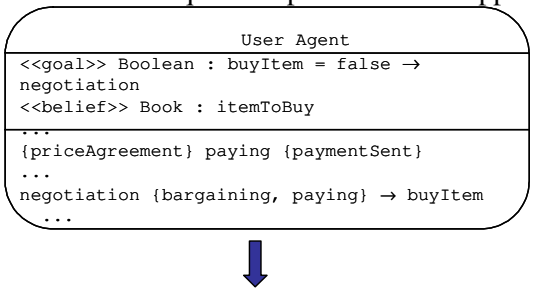
```
-- Relating actions to the plan
createActions : uml!Method (
        specification <- operationActions,
        body <- c.actions -> iterate (e; body : String = '' | body+'
                Action newAction = null;
                actions = new Vector();
                newAction = new '+e.name+'();
                actions.add(newAction);')
),
operationActions : uml!Operation (
        isAbstract <- false,
        specification <- 'void createActions()'
),
---------------------------------
-- Relating goals to the plan
createGoals : uml!Method (
        specification <- operationGoals,
        body <- 'Goal newGoal = null;
                newGoal= new
Goal('+c.goal.value+','+c.goal.valueType+','+c.goal.goalType+');
                goals.add(newGoal);'
),
operationGoals : uml!Operation (
        isAbstract <- false,
        specification <- 'void createGoals()'
)}
```

Applying the Transformations in a Simple Example

The UML model generated by the transformation is a UML class diagram that contains the ASF framework classes and the classes related to the application that instantiate the framework. Since this paper does not concern the MAS-ML dynamic diagrams during the transformation, UML dynamic diagrams are not part of the target model and, therefore, details about the execution of the agents were not transformed. All application entities, properties and relationships modeled on the three MAS-ML structural diagrams are represented in the target UML class model. Figure 4 depicts the transformation of the agent class *UserAgent* modeled in a MAS-ML organization diagram into a set of three classes (and its corresponding methods) modeled in a UML class diagram instantiating ASF.

4 Embracing the Evolution of Applications and Techniques

The evolution of the applications' requirements is easily handled by our MDE approach. After changing the design models according to the updated requirements, the implementation models can be regenerated by using the same set of rules already available. Changes in the requirements of an application do not require adaptations of our approach.



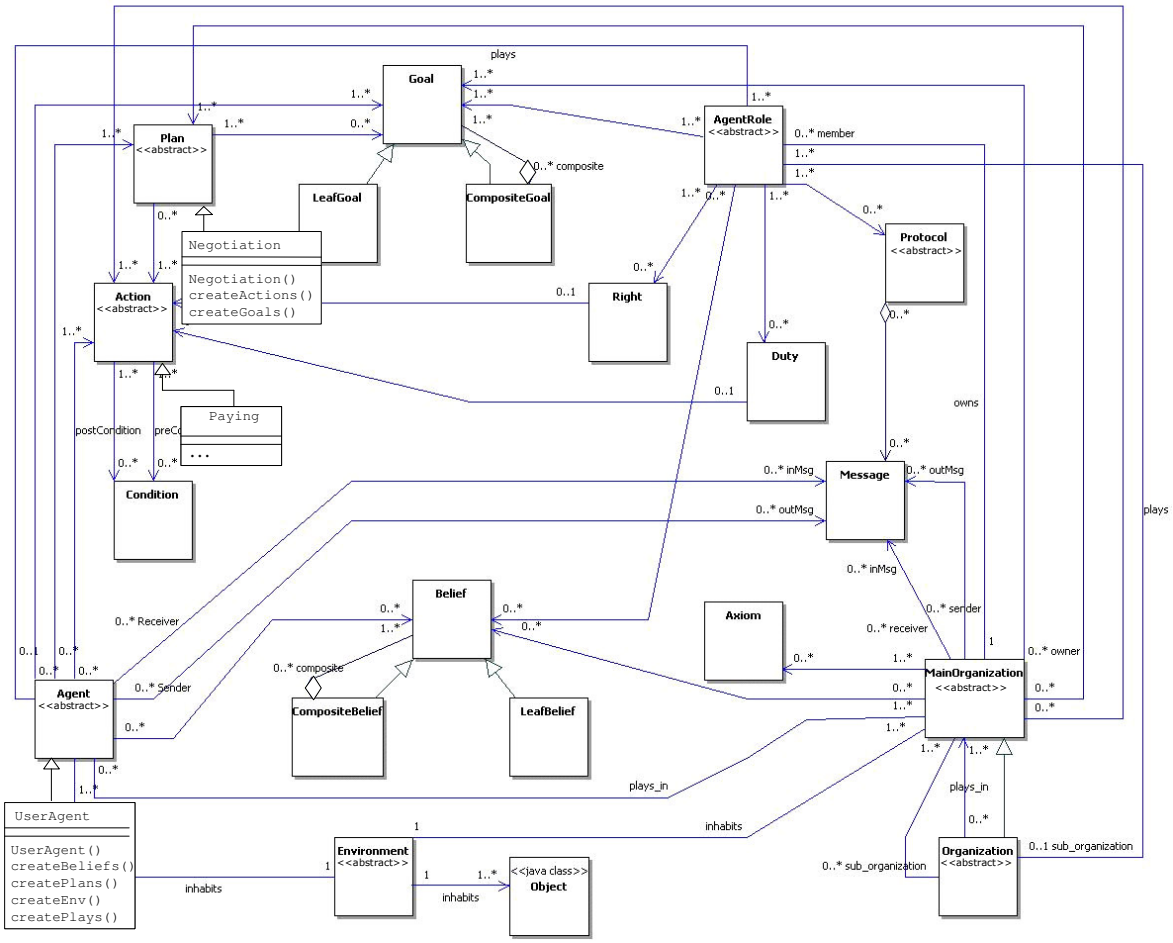


Figure 4. Transformation of a MAS-ML model into a UML model by instantiating ASF

Besides the evolution of applications, the techniques being used in our approach may also evolve. It may occur adaptations in the MAS-ML metamodel, the ASF framework and also in the UML metamodel. These three different technology adaptations influence the transformation rules. The evolution of the MAS-ML and UML metamodel influence the transformation rules since the rules are defined to receive well-formed models according to the source metamodels and to generate well-formed models according to the target metamodel. Adaptation of the ASF framework clearly influence the transformation rules since the generated target models are instances of the framework and, therefore, conforms to its specification. If the framework changes its instances also change.

Since the design and implementation models are low coupling, changes in the MAS-ML metamodel do not influence in the ASF specification or in the UML metamodel, and vice-versa. Besides, the rules were defined to try to minimize the effort of changing them due to technique evolution, as detailed below:

- *There is one rule for creating all the classes defined in ASF.* Therefore, if the ASF specification evolves, it will have a minor influence into the whole set of transformation rules. The rule that generates the ASF classes will need to be modified together with few

other rules, depending on the adaptation. For instance, if the abstract *Agent* class is modified it may be necessary to modify the *Agent* rule that generates the specializations of the abstract *Agent* class.

- *There is one rule for each entity type defined in MAS-ML.* If the MAS-ML metamodel evolves, it will be necessary to adapt the correspondent rule(s) that deal(s) with the modified part. Fortunately, each transformation rule deals with the transformation of one entity type defined in the MAS-ML metamodel. For instance, if the part of the metamodel that specifies the entity type *agent* is adapted, it will only be necessary to modify the *Agent* rule.
- *There is one rule for generating the plans, actions and protocols*. Since there is a need for creating OO classes for implementing plans, actions and protocols, we have defined a rule for transforming such properties. Thus, if an adaptation occurs in MAS-ML that affects how such properties are modeled or in ASF that affects how such properties are implemented, these rules must also be adapted.
- *It is easy to find out the part of the UML metamodel being used in the rules.* Each rule identifies the part of source model that the rule is able to transform and the part of the target model that it can generate. Such identification is done by point out the metaclasses of the source and target metamodels. Therefore, if the UML metamodel is modified, it will be necessary to search in all transformation rules the rules that deal with the modified part. For instance, if the specification of *Method* is changed, it will be necessary to search in all rules the part of the rules that generate methods, i.e., the part of the rules where the sentence *uml!Method* is stated.

5 Related Work

Although, some MAS methodologies such as Prometheus [15], Tropos [2] and MaSE [4] have not used an MDA approach, they have already proposed the mapping between the design models into implementation code and have also provided some tools for supporting both the design and the implementation of MAS. However, they do not clearly demonstrate the mapping from design models into code by presenting the rules used in the transformation. Therefore, it is extremely difficult to use the design models created by using the methodologies to generate code to another platform or framework that has not been addressed by them.

In addition, they do not separate the models into platform independent models and platform specific models. By using some of these methodologies, it is possible to describe platform specific details during the design of the application. In such cases, the high-level design models are platform dependent and, consequently, are not easily portable to any other platform.

Other authors have already used the MDA approach in other to define a MAS development process. Vallecillo et al [21] demonstrate the use of MDA to derive MAS low-level models from MAS high-level models. The authors propose to use the Tropos methodology and the Malaca platform [1] in the MDA approach. Malaca is platform where agents are defined based on the specification and reuse of software components. The high-level models created while using the Tropos methodology are transformed into low-level Malaca models. However, the transformation from the Tropos models into Malaca models is

not completely automated. It requires manual intervention. Moreover, such an approach does not deal with the transformation from Malaca models into code.

Novikay [12] analyzes how GR [3] based on the Tropos visual model can be related to MDA. The author interprets the MDA approach as a visual modeling activity where more abstract models are refined in more detailed models, using transformation techniques. This work covers only the requirement stage existent in Tropos. The difference between our approach and this approach is that ours contemplates the PIM, PSM and code stages.

In Kazakov et al. [10], the authors recommended a methodology based on a model-driven approach for the development of distributed mobile agent systems. They define a mobile agent conceptual model for distributed environments and describe a set of components, represented by a collection of intelligent mobile agents. While such an approach focuses on a specific application domain, our approach is a domain-independent development process.

6 Conclusion and Future Work

The MAS development process presented in this paper intends to provide an approach for modeling and implementing MAS by using MDE. We presented a *language translation* approach that is based on the translations between the source metamodel and the target metamodel. We have implemented a set of transformation rules by using the ATL transformation languages and to several MAS such as a supply chain management system [7][8] as well as web-based paper submission and reviewing system [5][23].

The proposed MDE based development process was illustrated by the use of MAS-ML and ASF. Our intention while using such techniques was to demonstrate how complex it is for transforming design models into implementation code due to the use of different paradigms while modeling the applications and while implementing them. Although MAS-ML and ASF are founded in the same agent's properties and characteristics, it is still not an easy task to manually instantiate ASF to implement MAS-ML design models. Therefore, the use of an (semi-)automatic transformer tool that could generate implementation code from design models is especially important while dealing with modeling language and platforms that do not share the same set of properties and characteristics. Although some times it may be very difficult to define transformation rules, once those rules are defined the implementation of any design model can easily be generated.

A prototyping developing tool [6] was created in order to demonstrate the feasibility of our approach. The tool allows the designers to graphically model MAS systems by using MAS-ML and to implement them while generating Java code by using the ASF framework. With the aim of enhancing the tool, several important improvements should be made. First, the transformer that generates code from MAS-ML models should also consider the MAS-ML dynamic diagrams. Second, the tool should make the visualization and also the modification of the UML models that represent the system implementation feasible. In addition, the tool should provide a model checker to analyze and verify the consistency of the different models (MAS-ML models and UML models).

References

[1] Amor, M.; Fuentes, L.; Troya, J. A Component-Based Approach for Interoperability Across FIPA-Compliant Plataforms. Cooperative Information Agents VII, LNAI 2782, p. 266-288. 2003.

[2] Bresciani, P. Tropos: An Agent-Oriented Software Development Methodology. Int. Journal of Autonomous Agents and Multi-Agents Systems, 8(3):203-236, 2004.
[3] Corradini, A., Montanari, U., Rossi, F., Ehrig, H., Heckel, R. and Lowe M. Algebraic Approaches to Graph Transformation. Handbook of Graph Grammars and Computing by Graph Transformation, 1997.
[4] DeLoach, S. A. Multiagent Systems Engineering: a Methodology and Language for Designing Agent Systems. In: Proc. of Agent Oriented Information Systems, 1999.
[5] DeLoach, S. A. Analysis and Design using MaSE and agentTool. In Proc. 12th Midwest Artificial Intelligence and Cognitive Science Conference, 2001.
[6] DeMaria, B.; Silva, V.; Chore, R.; Lucena, C. VisualAgent: A Software Development Environment for Multi-Agent Systems. In: Tool Track of Brazilian Symposium on Software Engineering, 2005.
[7] Fox, M. S.; Barbuceanu, M., Teigen, R. Agent-oriented Supply-chain Management. The International Journal of Flexible Manufacturing, v.12, p.165-188. 2000.
[8] Huget, M. Agent UML Class Diagrams Revisited. In: Proc. of Agent Technology and Software Engineering (AgeS), 2002.
[9] Jouault, F, and Kurtev, I. On the Architectural Alignment of ATL and QVT. In: Proc. of ACM Symposium on Applied Computing, model transformation track, Dijon, France, 2006.
[10] Kazakov, M., Abdulrab, H., Debarbouille, G. A Model Driven Approach for Design of Mobile Agent Systems for Concurrent Engineering: MAD4CE Project 2002.
[11] Kent, S. Model Driven Engineering. In Proceedings of Third Internations Conference of Integrated Formal Methods, Springer, LNCS 2335, pp. 286-298, 2002.
[12] Novikay, A. Model Driven Architecture approach in Tropos. Technical Report T04-06-03, Istituto Trentino di Cultura, 2004.
[13] Odell, J., Parunak, H. Bauer, B. Extending UML for Agents. In Proceedings of Agent-Oriented Information System Workshop at AAAI, pp. 3-17, 2000.
[14] OMG, UML: Unified Modeling Language Specification. Version 2.0. Available at: http://www.omg.org/uml/. Accessed in: 02/2005.
[15] Padgham, L, Winikoff, M. Prometheus: A Methodology for Developing Intelligent Agents, In Proc. of the 1st Int. Joint Conf. on Autonomous Agents and MAS, 2002.
[16] Pokahr, A. Braubach, L., Lamerdorf, W. Jadex: Implementing a BDI-Infrastructure for Jade Agents. Research of Innovation, 3(3):76-85, 2004.
[17] Silva, V., Lucena, C. From a Conceptual Framework for Agents and Objects to a Multi-Agent System Modeling Language. Journal of Autonomous Agents and MAS, Kluwer, 9(1-2), 2004.
[18] Silva, V., Cortes, M., Lucena, C. An Object-Oriented Framework for Implementing Agent Societies. Technical Report MCC32/04, PUC-Rio. Rio de Janeiro, Brazil, 2004.
[19] Silva, V., Choren, R., Lucena, C. Using the MAS-ML to Model a Multi-Agent System. Software Engineering for Large-Scale Multi-Agent Systems II, Springer, 2004.
[20] Sycara, K., Paolucci, M., Van Velsen, M., Criampapa, J. The Retsina MAS Infrastructure. Special joint issue of Autonomous Agents and MAS, 7(1-2):29-48, 2003.
[21] Vallecillo, A., Amor, M., Fuentes, L. Bridging the Gap Between Agent-Oriented Design and Implementation Using MDA. Autonomous Agents and MAS Workshop, pp.93-108, 2004.
[22] Wagner, G. The Agent-Object-Relationship Metamodel. In: Second International Symposium: From Agent Theory to Agent Implementation, 2000.

[23] Zambonelli, F.; Parunak, H. From design to intention: signs of a revolution. In: Proc. of the 1st Int. Conference on Autonomous Agents and MAS, pp. 455-456. 2002.