This item is the archived peer-reviewed author-version of:

On the use of a domain-specific modeling language in the development of multiagent systems

Reference:
Challenger Moharram, Demirkol Sebla, Getir Sinem, Mernik Marjan, Kardas Geylani, Kosar Tomaž.- On the use of a domain-specific modeling language in the development of multiagent systems
Engineering applications of artificial intelligence - ISSN 0952-1976 - 28(2014), p. 111-141
Full text (Publisher's DOI): https://doi.org/10.1016/J.ENGAPPAI.2013.11.012
To cite this reference: https://hdl.handle.net/10067/1709260151162165141

uantwerpen.be
Institutional repository IRUA

On the use of a Domain-Specific Modeling Language in the Development of Multiagent Systems

Moharram Challenger[1,a], Sebla Demirkol[1,b], Sinem Getir[1,c], Tomaž Kosar[2,d], Geylani Kardas[1,e], Marjan Mernik[2,f]

[a]moharram.challenger@mail.ege.edu.tr, [b]sebla.demirkol@ege.edu.tr, [c]sinem.getir@ege.edu.tr, [d]tomaz.kosar@uni-mb.si, [e]geylani.kardas@ege.edu.tr, [f]marjan.mernik@uni-mb.si

[1] International Computer Institute, Ege University, Bornova, 35100 Izmir, Turkey.
[2]Faculty of Electrical Engineering and Computer Science, University of Maribor, Smetanova 17, 2000 Maribor, Slovenia.

Abstract

The study of Multiagent Systems (MASs) focuses on those systems in which many intelligent agents interact with each other. The agents are considered to be autonomous entities which contain intelligence that serves for solving their selfish or common problems, and to achieve certain goals. However, the autonomous, responsive, and proactive natures of agents make the development of agent-based software systems more complex than other software systems. Furthermore, the design and implementation of a MAS may become even more complex and difficult to implement when considering new requirements and interactions for new agent environments like the Semantic Web. We believe that both domain-specific modeling and the use of a domain-specific modeling language (DSML) may provide the required abstraction, and hence support a more fruitful methodology for the development of MASs. In this paper, we first introduce a DSML for MASs with both its syntax and semantics definitions and then show how the language and its graphical tools can be used during model-driven development of real MASs. In addition to the classical viewpoints of a MAS, the proposed DSML includes new viewpoints which specifically support the development of software agents working within the Semantic Web environment. The practical use of the DSML is exemplified with a case study on the development of an agent-based expert finding system.

Keywords

Agent, Multiagent System, Model Driven Development, Domain-specific Modeling Language, Metamodel, Semantic Web.

1. Introduction

The development of intelligent software agents keeps its emphasis on both artificial intelligence and software engineering research areas. In its widely-accepted definition, an agent is an encapsulated computer system (mostly a software system) situated within a certain environment, and which is capable of flexible autonomous action within this environment in order to meet its design objectives (Wooldridge and Jennings, 1995; Bădică et al., 2011). These autonomous, reactive, and proactive agents also have social ability and can interact with other agents and humans in order to solve their own problems. They may also behave in a cooperative manner and collaborate with other agents for solving common problems. In order to perform their tasks and interact with each other, intelligent agents constitute systems called Multiagent Systems (MASs).

The implementation of agent systems is naturally a complex task when considering the aforementioned characteristics. In addition, the internal agent behavior model and any interaction within the agent organizations become even more complex and hard to implement when new requirements and interactions for new agent environments like the Semantic Web (Berners-Lee et al., 2001; Shadbolt et al., 2006) are taken into account.

The Semantic Web improves the current World Wide Web such that the web page content can be organized in a more structured way by tailoring it towards the specific needs of end-users. The web can be interpreted with ontologies (Berners-Lee et al., 2001) that help machines to understand web content. It is apparent that the interpretation in question can be realized by autonomous computational entities, like agents, to handle the semantic content on behalf of their human users. Software agents

can be used to collect Web content from diverse sources, process the information, and exchange the results.
In addition, autonomous agents can also evaluate semantic data and collaborate with semantically-defined entities of the Semantic Web, like semantic web services, by using content languages (Kardas et al., 2009a). Semantic web services can be simply defined as the web services with semantic interfaces to be discovered and executed (Sycara et al., 2003). In order to support the semantic interoperability and automatic composition of web services, the capabilities of web services are defined in service ontologies that provide the required semantic interfaces. Such interfaces of the semantic web services can be discovered by software agents and then these agents may interact with those services to complete their tasks. Engagements and invocations of a semantic web service are also performed according to the service's semantic protocol definitions. For instance, the dynamic composition of heterogeneous services for the optimal selection of service providers can be achieved by employing agents and ontologies, as proposed in (Bădică et al., 2012).

However, agent interactions with semantic web services add further complexities when designing and implementing MASs. Therefore, it is natural that methodologies are being applied to master the problem of defining such complex systems. One of the possible alternatives is represented by domain-specific languages (DSLs) (van Deursen et al., 2000; Mernik et al., 2005; Varanda-Pereira et al., 2008; Fowler, 2011) that have notations and constructs tailored towards a particular application domain (e.g. MAS). The end-users of DSLs have knowledge from the observed problem domain (Sprinkle et al., 2009), but they usually have little programming experience. Domain-specific modeling languages (DSMLs) further raise the abstraction level, expressiveness, and ease of use. The main artifacts of DSML are models instead of software codes and they are usually specified in a visual manner (Schmidt, 2006; Gray et al., 2007). Note that graphical notation for DSMLs is not mandatory and textual DSMLs also exist (Sánchez Cuadrado and Garcia Molina, 2007; Mernik, 2013), although the majority of DSMLs do indeed use visual notation.

DSMLs are used in Domain-specific Modeling (DSM), a software engineering methodology which is a particular instance of model-driven engineering (MDE) (Schmidt, 2006). A DSML's graphical syntax offers benefits, like easier design, when modeling within certain domains. The development of DSML is usually driven by language model definition (Strembeck and Zdun, 2009). That is, concepts and abstractions from the domain which need to be defined in order to reflect the target domain (language model). Then, relations between the language concepts need to be defined. Both form an abstract syntax of modeling language. Usually, a language model is defined with a metamodel. The additional parts of a language model are constraints that define those semantics which cannot be defined using the metamodel. Constraints are usually defined in a certain dedicated language (e.g. Object Constraint Language (OCL) (OMG, 2012)). Domain abstractions and relations need to be presented within a concrete syntax and serve as a modeling block within the end-user's modeling environment. This modeling environment can be generated automatically if dedicated software is used (e.g. MetaEdit+ (MetaCase, 1995) and Eclipse GMF (The Eclipse Foundation, 2006)), otherwise, modeling editor must be provided by hand (Kos et al., 2011). Then, the model transformations need to be defined in order to call the domain framework, which is a platform that provides the functions for implementing the semantics of DSMLs within a specific environment. Usually, the semantics is given by translational semantics (Bryant et al., 2011).

We are convinced that both DSM and the use of a DSML may provide the required abstraction and support a more fruitful methodology for the development of MASs. Hence, in this paper, we first introduce a DSML for MASs with both its syntax and semantics definitions, and then show how the language and its graphical tools can be used during the model-driven development of real MASs. The domain of our study is "Semantic Web enabled MASs" in which autonomous agents can evaluate semantic data and collaborate with semantically-defined entities of the Semantic Web, like Semantic Web Services. In order to support MAS experts when programming their own systems, and to be able to fine-tune them visually, our DSML covers all aspects of an agent system from the internal view of a single agent to the complex MAS organization. In addition to these classical viewpoints of a MAS, the proposed DSML also includes new viewpoints which specifically pave the way for the development of software agents working on the Semantic Web environment. We refer to this DSML

as a Semantic web Enabled Agent Modeling Language (SEA_ML). Our concrete motivation for SEA_ML is to enable domain experts to model their own MASs on the Semantic Web without considering the limitations of using existing MAS development frameworks (e.g. JADE (Bellifemine et al., 2001), JADEX (Pokahr et al., 2005) or JACK (Howden et al., 2001)).

The remainder of the paper is organized as follows: Sections 2 and 3 discuss the abstract and concrete syntaxes of SEA_ML. The model-to-model and model-to-text transformations which are the building blocks of SEA_ML's semantics are discussed in Sections 4 and 5, respectively. Section 6 includes a practical case study on the development of a MAS by using SEA_ML. Related work is given in Section 7. Finally, Section 8 concludes the paper and states the future work.

2. Abstract Syntax

The abstract syntax of a DSML describes the concepts and their relations to other concepts without any consideration of meaning. In other words, the abstract syntax of a language describes the vocabulary of concepts provided by the language and how they may be combined to form models or programs (Clark et al., 2004). In terms of Model Driven Development (MDD), the abstract syntax is described by a metamodel which defines what the models should look like. Hence, in this section, we discuss the metamodel that constitutes the SEA_ML's abstract syntax.

In a Semantic Web enabled MAS, software agents can gather Web contents from various resources, process the information, exchange the results, and negotiate with other agents. Within the context of these MASs, autonomous agents can evaluate semantic information and work together with semantically-defined entities like semantic web services using content languages. The work in (Kardas et al., 2009a) defined a conceptual architecture that contained an overview of such a MAS. Originating from this architecture, Kardas et al. also provided a metamodel which covers those meta-elements that belong to Semantic Web enabled MASs. Reengineering of that metamodel enabled us to form the platform independent metamodel (PIMM) which represents the abstract syntax of SEA_ML. Resulting metamodel focuses on both modeling the internal agent architecture and MAS organization. When the SEA_ML's metamodel is overviewed, one can detect that one of the main elements within the metamodel is the Semantic Web Agent (SWA). A SWA works within a Semantic Web Organization (SWO) and can interact with various services. Another meta-element is the Semantic Web Service (SWS) which represents a web service (except agent service) defined semantically. Also, a SWS is composed of one or more Web Service entities. The corresponding services must have a semantic interface that is going to be used by the platform's agents.

Agents need to apply a service registry for the purpose of discovering service capabilities. Therefore, the SEA_ML's metamodel has another meta-element called the Semantic Service Matchmaker Agent (SSMatchmakerAgent) which is a SWA extension. This meta-element represents matchmaker agents which store the SWS' capabilities list in a MAS and compare it with the service capabilities required by the other agents, in order to match them. SWA uses ontologies for managing internal information and making inferences based on its facts within the scope of its roles. A SWA can associate with one or more Roles and change its Role over time.

The Object Management Group's (OMG) Ontology Definition Metamodel (ODM) has been plugged into the SEA_ML's metamodel in order to help in the definition of ontological concepts. Moreover, in addition to the reactive architecture, SEA_ML abstract syntax also supports the modeling of Belief-Desire-Intention (BDI) Agents (Rao and Georgeff, 1995) with new meta-entities and their relations, which are not covered in (Kardas et al., 2009a).

In order to provide clear understanding and efficient use, the SEA_ML's metamodel is divided into eight viewpoints each describing different aspects of the Semantic Web enabled MASs. These viewpoints include MAS, Agent Internal, Plan, Role, Interaction, Environment, Agent-SWS Interaction, and the Ontology viewpoints. The diagrams for each of the partial metamodels of these viewpoints (Figures 1 to 8) are the Ecore diagrams extracted automatically from the metamodels

defined in the Ecore files. In fact, Ecore is based on EMOF (The Eclipse Foundation, 2006) and formally specifies the metamodels. These metamodels constitute the abstract syntax within our study.

All of the SEA_ML viewpoints are discussed in detail in the following subsections. In each viewpoint's diagram, the elements filled-in with light gray come from those other viewpoints which are shown at the top or bottom of the element using “<<” and “>>” characters. In other words, these elements are common elements amongst viewpoints, and tailor them to each other. For example, in the MAS viewpoint, the Role meta-element comes from <<Role Viewpoint>> (see Figure 1). Furthermore, the main elements of each viewpoint are depicted using darker borders. Taking the MAS viewpoint into consideration, SWO is the main element of the viewpoint and has a darker border than the other elements of the viewpoint (see Figure 1).

2.1 MAS Viewpoint

The SEA_ML's MAS viewpoint deals solely with the construction of a MAS as an overall aspect of the metamodel. It includes the main blocks which compose the complex system as an organization (Figure 1). The SWO entity of the SEA_ML metamodel is a composition of those SWAs having similar goals or duties. An agent cooperates with one or more agents inside an organization and may also reside in more than one organization over time. Moreover, a SWO can include several agents at any time and each organization can be composed of several sub-organizations recursively. Each organization interacts with an Environment which by itself includes all of the resources, services, and non-Agent concepts like a database (as discussed in subsection 2.6). The SWAs use the resources of the SWO within which they work. Also, a SWO can have one or more Roles that represent the organization's aims like trading, education, medical issues, and so on.

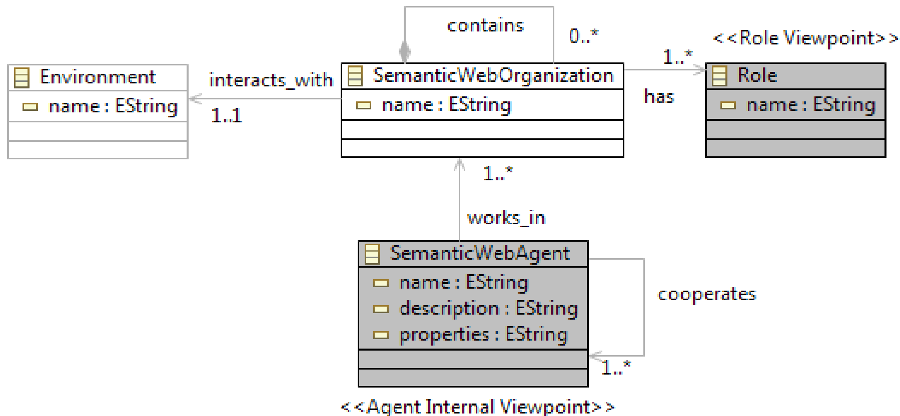

Figure 1: MAS viewpoint.

2.2 Agent Internal Viewpoint

This viewpoint, as part of whole metamodel, focuses on the internal structure of every agent within a MAS organization. As can be seen in Figure 2, the Semantic Web Agent in the SEA_ML abstract syntax stands for each agent which is a member of Semantic Web enabled MAS. A Semantic Web Agent is an autonomous entity which is capable of interacting with both the other agents and the semantic web services, within the environment. They can play roles and use ontologies to maintain their internal knowledge and infer about the environment based on the known facts. Semantic Web Agents can be associated with more than one *Role* (multiple classifications) and can change these roles over time (dynamic classification). By taking different types of roles into consideration, an agent can play a Manager role, a Broker role, or a Customer Role. An agent can only have one state (*Agent State*) at a time, e.g. waiting state in which the agent is passive and waiting for another agent or resource. Similarly, it can be active whilst doing the internal or external processes. Therefore, it helps an agent to decide about communication with another agent by considering its state. An agent can also have a type (*Agent Type*) during its life based on the application in which it is going to take part, such

as buyer agent/shopping bot, user/personal agent, monitoring-and-surveillance agent, or data mining agent (Haag et al., 2004). In addition, it can stop working in one organization and start to work in another.

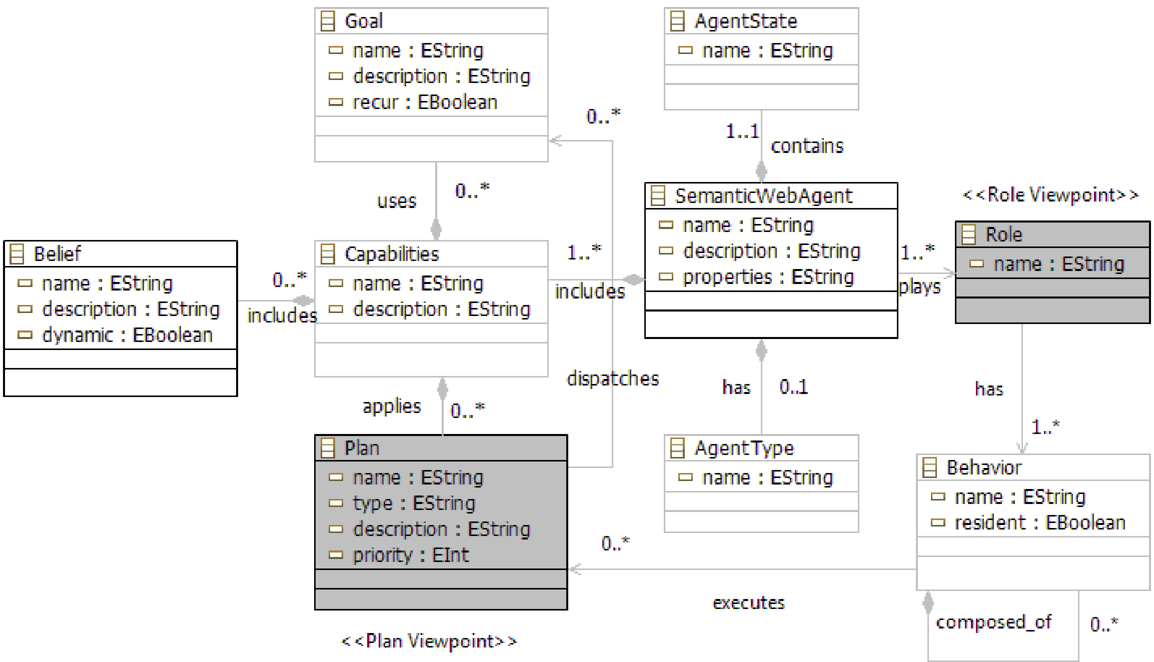

Figure 2: Agent Internal viewpoint.

As previously mentioned, SEA_ML's abstract syntax supports both reactive and BDI agents. While BDI agents are supported by providing Beliefs, Plans, and Goals, the reactive agents are also supported by Behaviors and a nested structure for them. Using this nested structure, the modeling of different compositions of behaviors is possible. As discussed in (Vidal et al., 2001), a reactive agent does not maintain information about the state of its environment but simply reacts to current perceptions. In fact, "it is not much more than an automaton that receives input, processes it and produces an output" (Ferber, 1999). On the other hand, in BDI (Rao and Georgeff, 1995) architecture, an agent decides on which *Goals* to achieve and how to achieve them. Beliefs represent the information an agent has about its surroundings, whilst Desires correspond to those things an agent would like to achieve. Intentions, which are the deliberative attitudes of agents, include the agent's planning mechanism in order to achieve goals. Taking BDI agents into consideration, we propose an entity called *Capabilities* which includes each agent's *Goals*, *Plans* and *Beliefs* about the surroundings. In other words, a capability concept is in fact an agent without reasoning power. It also provides reusability for *Goals*, *Plans* and *Beliefs*. Each *Behavior* can execute some *Plans* and as a result of this, *Goals* will be achieved. *Roles* are also directly connected to the *Behaviors* by taking into account the reactive agents.

2.3 Plan Viewpoint

As the name already describes, the Plan viewpoint defines the internal structure of an agent's plans. When an agent applies a *Plan*, it executes its *Tasks* which are composed of the more atomic elements called *Actions*. Since *Actions* are atomic, they have executable structures from which *Send* and *Receive* elements are extended. These action types are connected with a *Message* entity as illustrated in Figure 3. Sending a message to another agent or querying an ontology are two examples of Action.

A Plan can represent any big process like registration. So, the tasks of a plan make it possible to divide a Plan's duty into several parts. Tasks may need to have subtasks. For this purpose, we have provided one more level of break-down for the Tasks by Actions. This supports the hierarchy for big

processes to have proper granularity at each level. If a developer needs more breaking-down, he/she can use functions or methods in the target language.

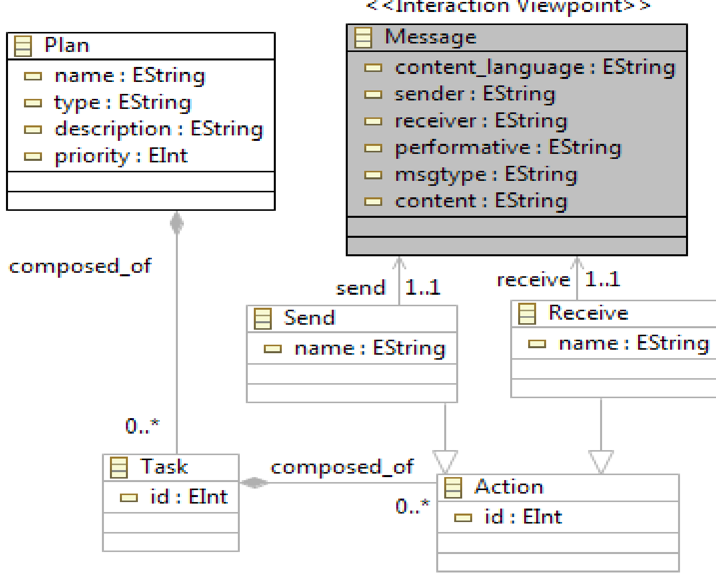

Figure 3: Plan viewpoint.

2.4 Role Viewpoint

SWAs and SWOs (as a whole) can play roles and use ontologies to maintain their internal knowledge, and infer about the environment based on the known facts. As mentioned in subsection 2.2, agents can also use several roles and can alter these roles over time. A *Role* is a general model entity and should be specialized in the metamodel according to architectural and domain tasks. An *ArchitectureRole* defines the mandatory roles for a Semantic Web enabled MAS (e.g. *RegistrationRole* and *OntologyMediatorRole*) which should be played with at least one agent inside the platform regardless of the organization. On the other hand, a *DomainRole* depends completely on the requirements and task definitions of a specific SWO created for a specific business domain. Since a Role can have various duties, it can have different interactions with different agents.

As can be seen in Figure 4, we cover Role types and their relations, as required within the whole metamodel. For example, the *RegistrationRole*, which is a type of *ArchitectureRole*, is a crucial role type for semantic service registration. Specifically, this role advertises the Semantic Web Services. Finally, an *OntologyMediatorRole* can be used with an agent to handle ontologies.

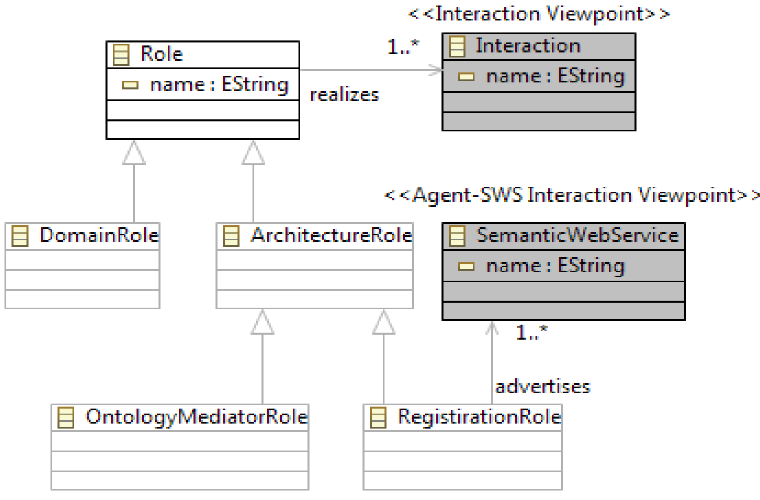

Figure 4: Role viewpoint.

2.5 Interaction Viewpoint

This viewpoint focuses on agent communications and interactions in a MAS, and defines entities and relations such as *Interaction*, *Message*, and *MessageSequence*, see Figure 5. Agents interact with each other based on their social abilities. Each interaction, by itself, consists of some Message submissions each of which should have a message type (msgtype) such as inform, request, or acknowledgement. Specifically, each communication between the initiator and the participating agents can be modeled with Messages which can also have performative properties (e.g. inform, query, or propose) compatible with the IEEE Foundation of Intelligent Physical Agents (FIPA) standard (FIPA, 2002a). The content language property of the Message entity is used for communication between agents and can be one of the communication languages like Knowledge Query and Manipulation Language (KQML) (Finin et al., 1997) or FIPA Agent Communication Language (ACL) (FIPA, 2002b). The Interaction element extends the *FIPAContractNet* element. The FIPAContractNet represents the IEEE FIPA's specification for the interactions of agents, which apply the well-known "Contract Net Protocol" (CNP) (Smith, 1980). In addition, each Interaction should have a MessageSequence to control the communication flow. Communication of the distributed agents can be handled by a sequence diagram or an activity diagram using this entity.

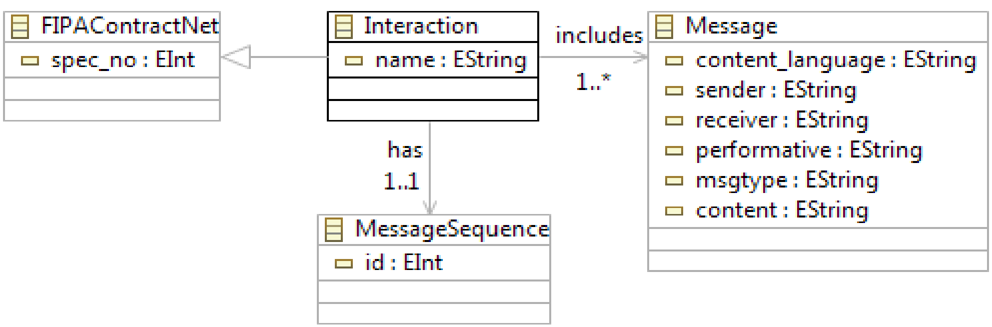

Figure 5: Interaction viewpoint.

2.6 Environment Viewpoint

The Environment viewpoint, Figure 6, focuses on the relations between agents and to what they access. The *Environment,* in which agents reside, contains all non-agent *Resource*s (e.g. database, network device), *Fact*s and *Service*s. Each service may be a web service or another service with predefined invocation protocol in real-life implementation. Facts are environment-based which means they can change over time, in case the Environment has new knowledge from different resources.

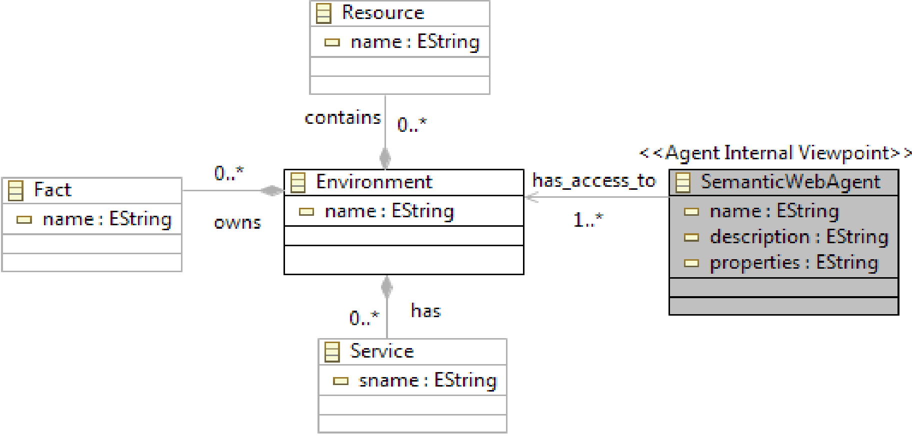

Figure 6: Environment viewpoint.

It is worth noticing the difference between an Environment's fact and an agent's belief. For example, in an Environment, the knowledge about the weather can be "It is 30°C". An agent can take this information to its Belief-base. Later, the fact "It is 30 °C" in the Environment can be altered to "It is 15 °C". In this case, if the agent does not update its knowledge, its fact will not change. This means, whilst the fact is changing, the belief can remain the same, and as a result the agent's knowledge can be different and inconsistent in regard to the real world's facts.

2.7 Agent-SWS Interaction Viewpoint

This viewpoint of the SEA_ML metamodel models the interaction between the agents and SWSs. The concepts and their relations for appropriate service discovery, agreement with the selected service and execution of the service are all defined within this viewpoint. Furthermore, the internal structure of SWS is modeled inside this viewpoint.

We consider MASs and SWSs as two standalone systems, which can however interact with each other to realize automatic service discovery, negotiation, and execution. The agents are able to perform tasks automatically (interaction with service profile, process model, and grounding) and locate related information on behalf of the human user. Our main goal is to bridge the required communication between software agents and the semantic web services. Although presenting the agent's own services in the semantic web services form is not mainly targeted; this is, however, also possible in our system. In order to do this a developer can model an agent's service as a SWS and the tool will generate the required documents for this agent service, which conform to SWS advertisement and utilization specifications.

When considering the decision making duties of agents, *SemanticWebAgent*s apply *Plan*s for performing their tasks. In order to dynamically discover the desired services, negotiate with them and execute the agreed one, the *SemanticWebAgent* entity owns three extensions of the Plan entity in the SEA_ML metamodel (Figure 7). *Semantic Service Finder Plan (SS_FinderPlan)* is a Plan in which automatic discovery of the candidate semantic web services takes place with the help of the *SSMatchmakerAgent*. *Semantic Service Agreement Plan (SS_AgreementPlan)* deals with the negotiations on the Quality of Service (QoS) metrics of the service (e.g. service execution cost, running time, and location) and the agreement settlement. After service discovery and negotiation, the agent applies the *Semantic Service Executor Plan (SS_ExecutorPlan)* to invoke appropriate semantic web services. As discussed earlier, Semantic Service Matchmaker Agents (shown as *SSMatchmakerAgent* in Figure 7) represent a service registry for agents to discover services according to their capabilities. In addition, a *Semantic Service Register Plan (SS_RegisterPlan)* can be applied by a SSMatchmakerAgent to register a new SWS. Hence, by interacting with a *SSMatchmakerAgent*, a *SemanticWebAgent* can apply its SS_*FinderPlan* and automatically select some interfaces of *SemanticWebServices* when considering QoSs and then dynamically negotiate with them using the *SS_AgreementPlan* for realizing the agreement on a service.

Semantic web service modeling approaches, e.g. OWL-S (Martin et al., 2004), mostly describe services using three semantic documents: Service Interface, Process Model, and Physical Grounding. Service Interface is the capability representation of the service in which service inputs, outputs and any other necessary service descriptions are listed. Process Model describes the internal composition and execution dynamics of the service. Finally, Physical Grounding defines the invocation protocol of the web service. These Semantic Web Service components are given within our metamodel as *Interface*, *Process* and *Grounding* entities, respectively. The *Input*, *Output*, *Precondition* and *Effect* (a.k.a. IOPE (Martin et al., 2004)) definitions used by these Semantic Web Service components are also defined. The metamodel imports the OWLClass meta-entity (shown as *ODMOWLClass* in Figure 7) from the OMG's ODM (OMG, 2009) as the base class for the semantic properties (mainly IOPE) of the semantic web services. Since the operational part of today's semantic services is mostly a web service, the *WebService* concept is also included within the metamodel and associated with the grounding mechanism.

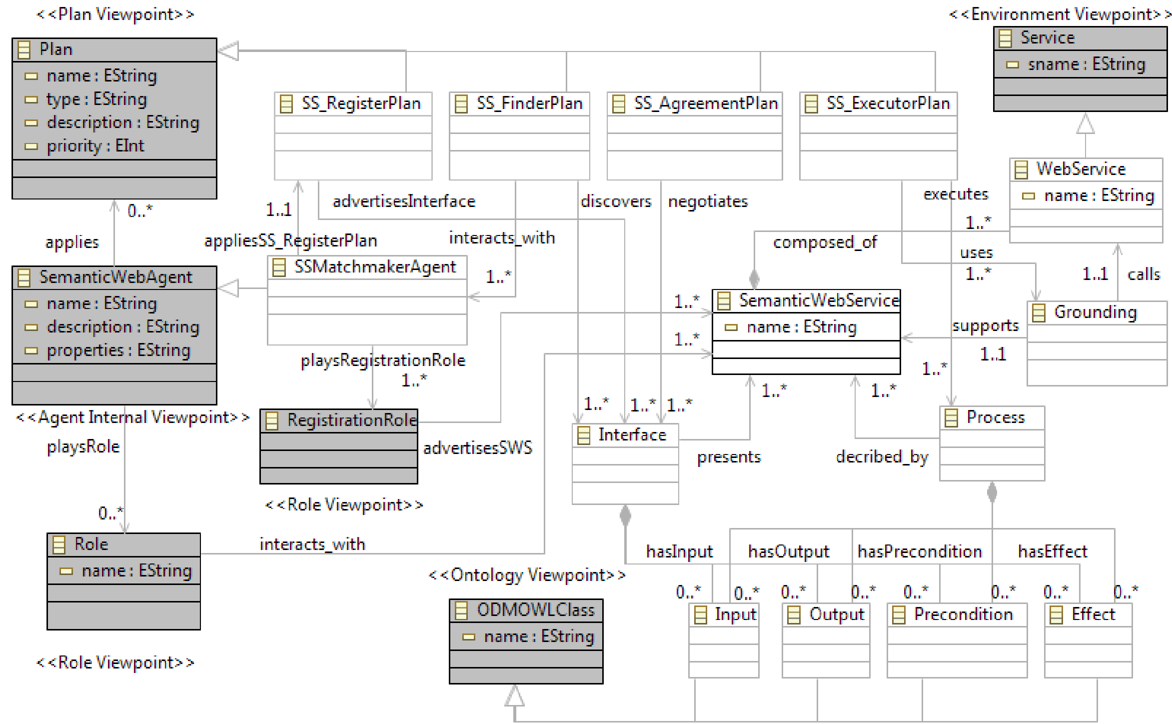

Figure 7: Agent-SWS Interaction viewpoint.

2.8 Ontology Viewpoint

A MAS Organization on the Semantic Web is inconceivable without ontologies. An ontology represents any information gathering and reasoning resource for MAS members. The ontology viewpoint brings all ontology sets and ontological concepts together. The ODM OWL Ontology from OMG's ODM (OMG, 2009) is a standard for all of our ontology sets such as *Role*, *Organization*, and *Service* Ontologies (Figure 8). According to this viewpoint, all the ontologies are known by their related elements. A collection of ontologies creates a knowledgebase of the MAS that provides domain context. These ontologies are represented in SEA_ML models as *OrganizationOntology* instances. Inside a domain role, an agent uses a *RoleOntology* which is defined for the related agent role concepts and their relations. The semantic interfaces and capabilities of Semantic Web Services are described according to *ServiceOntologies*. Finally, for the Semantic Web environment, each fact or an agent's belief is an ontological entity and they are modeled as an extension of the *ODM OWL Statement* from ODM.

3. Concrete Syntax

Whilst the specification of abstract syntax includes those concepts that are represented in the language and the relationships between those concepts, concrete syntax definition provides a mapping between meta-elements and their representations for models. In fact, the concrete syntax is the set of notations which facilitates the presentation and construction of the language. This section discusses graphical concrete syntax which maps the abstract syntax elements of SEA_ML to their graphical notations.

There are various tools which provide visual modeling environments for the development of concrete syntaxes of DSMLs. One of them is Graphical Modeling Environment (GME) (GME, 2001) which is based on a set of built-in generic concepts. It is also extensible and can be used for the GPLs (General Purpose Languages) such as C++, Visual Basic, C#, and Python. It has various concepts for building large-scale and complex models which are hierarchy, multiple aspects, sets, references, and explicit constraints. It is easy to use but does not have detailed modeling as much as in Eclipse GMF (The Eclipse Foundation, 2006).

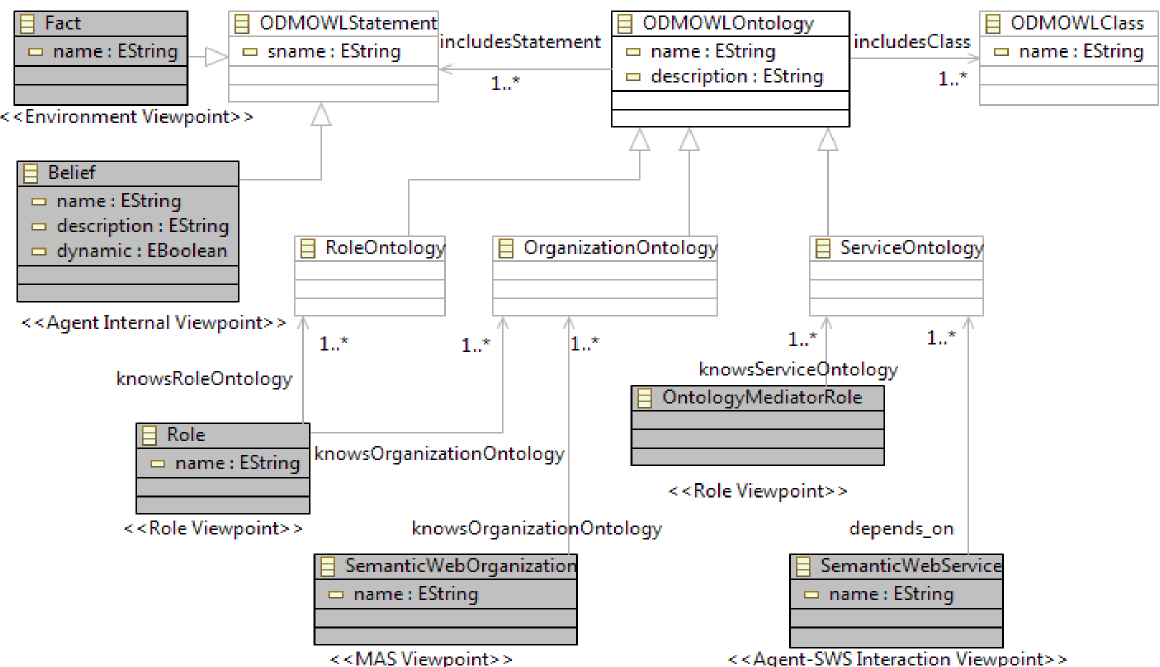

Figure 8: Ontology viewpoint.

Another tool, Freemind (Freemind, 2002), supplies a user friendly interface. The tool has the ability to keep track of projects and provides visual models of the test designs. It also supplies essay writing and brainstorming. However, it is not specially designed for modeling.

MetaEdit+ (MetaCase, 1995) is another widely-known tool. It is an integrated tool which includes metamodeling and modeling environments for a single user or multi-users during the tool evolution. MetaEdit+ claims that it provides easy usage for developers, and is also based on a strong background with the support of the MetaCase Company. However, it is neither an open source nor free.

Microsoft also released its first DSL tool in 2005 (Microsoft, 2005) which is part of Visual Studio and works only on Windows. It has a limited concrete syntax. For example, the symbols can only have a single geometrical figure (Kelly and Tolvanen, 2008). Similar to MetaEdit+, this tool is not an open source and free.

On the other hand, Eclipse GMF (The Eclipse Foundation, 2006) is an elaborated modeling tool for developing DSMLs. It has various beneficial components whilst developing software models. The first component is Ecore which enables developers to define metamodels at the meta-meta level. The second and third components are “Tooling Definition” and “Graphical Definition” components which provide palette creation with its properties in the tool and graphical facilities for the concrete syntax elements, respectively. Finally, the GMF mapping component provides the mapping between meta-elements and graphical facilities. The new platform can be executed, by generating the tool’s code with this component. Those features of Eclipse GMF caused us to prefer it during the development of SEA_ML's concrete syntax and related set of graphical modeling tools (Getir et al., 2011).

After setting the graphical notations for abstract syntax meta-elements, we use Eclipse GMF to tie notations to the domain concepts in an Ecore file. The graphical notations for MAS, Agent Internal, Agent-SWS Interaction, and Ontology viewpoints are listed in Tables 1, 2, 3 and 4 respectively since these are the more important viewpoints of our metamodel and cover the majority of the notations. In the tables there are no notations for the superclass elements of the metamodel which will not be instantiated in the instance model. Also, the composition relation is modeled with compartments in their superior element, so, there are no graphical notations for them either. The left columns define the names of the meta-elements in abstract syntax and the right columns mark notations or icons in the syntax tool of SEA_ML.

Table 1: The concepts and their notations for the graphical concrete syntax elements of the MAS viewpoint.

Concept	Notation
Semantic Web Organization	
Environment	
Role	
Semantic Web Agent	

Table 2: The concepts and their notations for the graphical concrete syntax elements of the Agent-Internal viewpoint.

Concept	Notation
Belief	
Goal	
Capabilities	
Plan	
Behavior	
Agent State	
Agent Type	
Role	
Semantic Web Agent	

Table 3: The concepts and their notations for the graphical concrete syntax elements of the Agent-SWS Interaction viewpoint.

Concept	Notation
Semantic Web Agent	
Semantic Service Matchmaker Agent	
Semantic Service Register Plan	R
Semantic Service Finder Plan	F
Semantic Service Agreement Plan	A
Semantic Service Executor Plan	E
Role	
Registration Role	
Semantic Web Service	
Process	P
Interface	I
Grounding	G

We decided that the elements of the same type or elements which inherit from the same element have similar icons and similar geometric notations. For example, similar icons and similar geometric notations are used for the SS_RegisterPlan, SS_FinderPlan, SS_AgreementPlan, and SS_ ExecutorPlan which inherit from the Plan entity by holding the same tint and ground and adding the first letter of the Plan's word in the icon, as can be seen in Table 3. These letters represent the types of Plan elements. Similarly, a combination of Ontology and other meta-elements is provided when their integration is needed. As an example, *RoleOntology* keeps the basic ontology background; it also holds the Role icon, as illustrated in Table 4.

The Ecore models are created in Eclipse, thus it can be seen as elements and relationships in the visual diagrams. These Ecore models are used during the process of developing GMF tools. During this process, icons are determined for both palettes and figures, the geometrics of icons are described and some constraint checkers are considered. The achieved artifacts are the graphical editors in which agent developers can design models for each viewpoint of the required MAS conforming to the concrete syntax of SEA_ML.

Table 4: The concepts and their notations for the graphical concrete syntax elements of the Ontology viewpoint.

Concept	Notation
Belief	
Fact	
Organization Ontology	
Role Ontology	
Service Ontology	
Semantic Web Organization	
Role	
Semantic Web Service	
Ontology Mediator Role	
ODMOWLClass	

SEA_ML's syntax tools can impose some restrictions/controls during the user's modeling. One part of these controls comes from the metamodel and the remaining originates from the GUI tool itself. So, these controls are divided and discussed in two parts, "Model Constraints" and "Graphical Tool Facilities" respectively.

Model Constraints:

The GMF-based constraints coming from the Ecore models of the SEA_ML are provided for any instance models of all viewpoints. These constraints can be classified as following:

Compartment constraint:
The composition relationship between the meta-elements in the Ecore is transformed to a relationship that one element contains the other. Two elements that do not have this kind of relationship cannot be modeled as a nested compartment. For instance, the Capabilities element is composed of Plans, Beliefs, and Goals in the metamodel according to the Agent Internal viewpoint. Thus, each Plan, Belief or Goal instance constitutes a compartment in Capabilities because of the composition

relationship between them. Contrarily, this nested modeling method cannot be used between a Role and a SWA.

Number of relationships constraint:
Due to one-to-one, one-to-many, many-to-many relationships in the Ecore, number of relationships are controlled between the elements in the instance models. For example, whilst a SWA can play more than one role in the Agent Internal viewpoint, it can only have one Agent Type.

Relationship source and destination constraint:
The direction of the relationship defines the source and destination of that relationship. This constraint is defined at the Ecore level. For example, the relationship between Plan and Goal cannot be created in the instance model where the direction is, in fact, from plan to goal.

Inheritance relationship constraint:
The defined inheritance relationships in SEA_ML syntax naturally force some constraints whilst creating the instance model. A subclass in an instance model will include all of the attributes and relationships of its super-class. The Agent-SWS Interaction viewpoint is the best example for this issue. Plan has relationships with other elements directly. It also has four subclasses and when they appear in the instance model, all of the relationships are inherited from the Plan element.

Since the Interaction viewpoint has basic communication and messaging relationships, it does not have a compartment structure. However, compartment constraint is used in all the other viewpoints which have the composition relationships between the meta-elements as the compartment's structure requires.

Graphical Tool Facilities:

While creating the model, in addition to the constraints coming from the metamodel, SEA_ML's modeling tool has some editorial constraints which help the tool user during his or her design phase. These constraints are listed below:

Double clicking on key elements – transition between viewpoints:
This constraint provides unity of the system by supplying a transition between different viewpoints' editors. The system has an overview of the MAS viewpoint. For example, in this editor when SWA and Role instances are dropped into the platform, the user can open the Role viewpoint editor by double clicking on the Role element. In the same way, by double clicking on the SWA instance, the Agent Internal viewpoint editor will be opened. When the user is oriented in this way, the system can be created step by step. Figure 9 illustrates the system integration and transitions between the viewpoints with the key elements in each viewpoint. However, the tool remains flexible whilst creating the diagram files. In other words, it is possible to create each of the viewpoint diagrams without any order, in case of the user's request. For example, the user can design an agent diagram for the instance model without designing the MAS diagram.

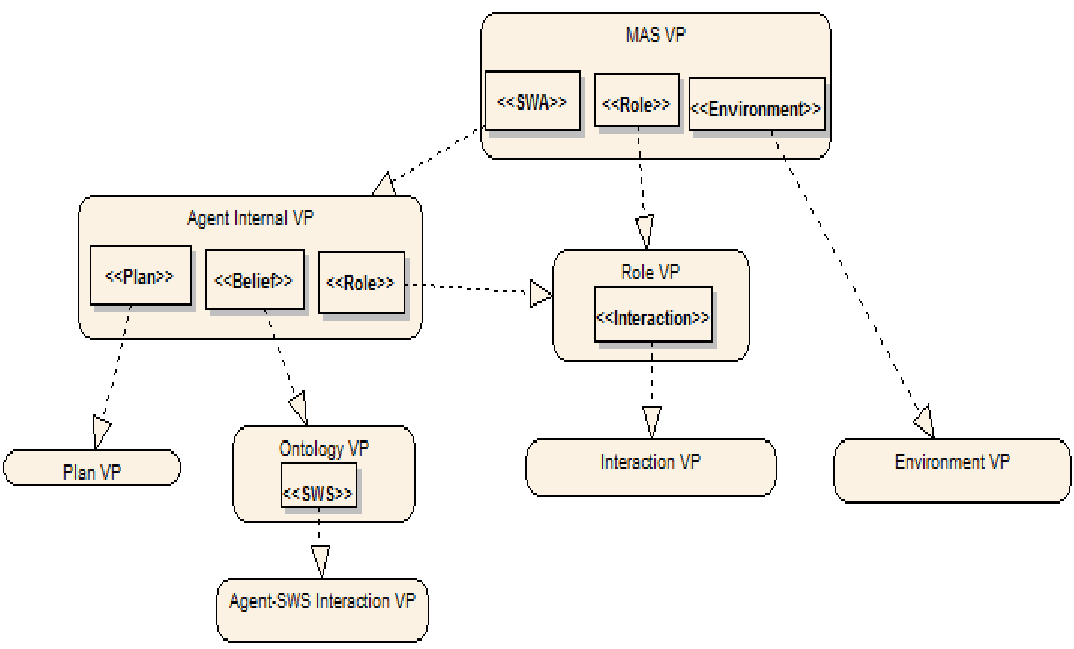

Figure 9: System integration and transitions between SEA_ML viewpoints.

Keeping previous entities and properties in all viewpoints' editors – Unification:
This property provides system unification for all the elements. The editor diagrams are created according to the viewpoints. On the other hand, system metamodel should be considered as a whole model. Therefore, any defined instance element should be saved in a list in order to be used uniquely during the whole modeling process. This is provided by having a tree structure that shows all those elements which can be included in any viewpoint diagram. For instance, when a SWA instance, created in an Agent Internal viewpoint editor, is needed in an Agent-SWS interaction diagram, it can be used by dragging and dropping from the unique tree of the instance elements.

Integrity of relationship-element constraint:
According to this constraint of the tool, when an element created in any instance model is removed, all of the relationships will be removed from the model. This leads to keep integrity of the whole model and the model remains consistent after this kind of modification.

Since some of the aforementioned constraints are fundamental, e.g. number of relationships, relationship's source and destination, and the integrity of relationship-element constraints, they exist in all of the viewpoints. Inheritance constraints can be seen in all of the SEA_ML viewpoints except the Organization viewpoint as it does not have any superclass relationship in its metamodel.

4. Model-to-Model Transformations: Operational Semantics for SEA_ML

It is not sufficient to complete a DSML definition by only specifying the notions and their representations. The complete definition requires that one provides semantics of language concepts in terms of other concepts the meanings of which are already established. Therefore, the abstract syntax of the SEA_ML is mapped into the metamodels of the existing agent platforms (such as JADE (Bellifemine et al., 2001), JACK (AOS, 2001) or JADEX (Pokahr et al., 2005)) and ontology languages (such as Web Ontology Language (OWL) (W3C, 2004)) which have well-defined, understood and executable semantics. Those mappings lead to model transformations that are applied on the SEA_ML model instances at runtime, in order to obtain their counterparts in real MAS infrastructures. Model-to-code transformations follow these model-to-model transformations and finally achieve executable software codes for exact MAS. The whole procedure is depicted in Figure 10.

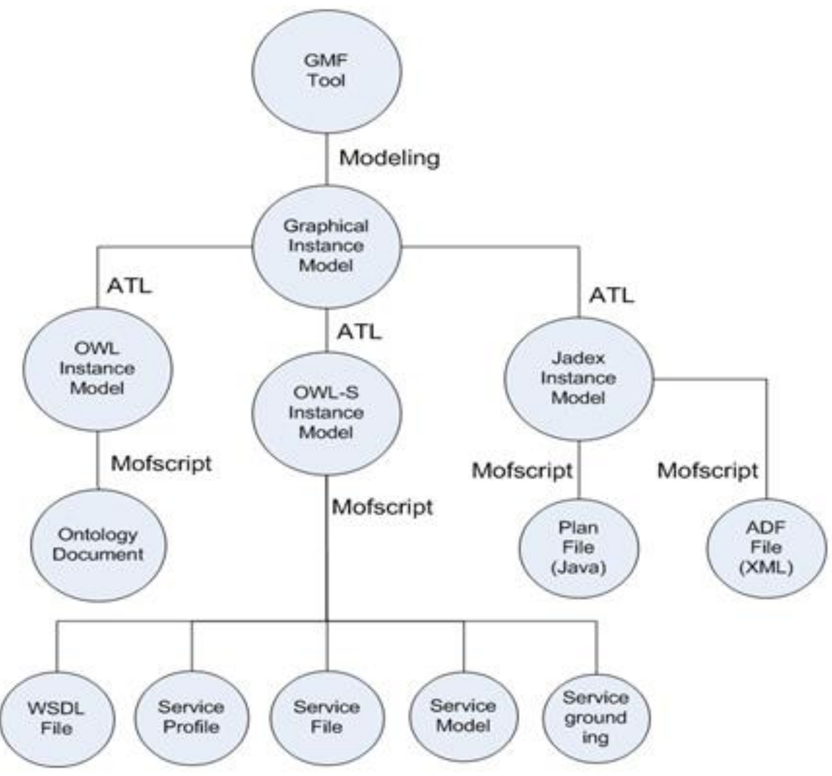

Figure 10: Applied procedure for the model driven development of MASs based on SEA_ML.

In this study, transformations between SEA_ML and the JADEX BDI agent framework (Pokahr et al., 2005, 2007) are defined and implemented for the production of agent software. Since it is possible to generate code from SEA_ML models to reactive agent languages like JADE (to its different types of behaviors), SEA_ML is not limited to BDI agents and can support reactive agents. Furthermore, semantic web components (e.g. the agent knowledgebases and semantic web services) of agent systems modeled according to SEA_ML are obtained via transformations from SEA_ML to OWL and OWL-S (Martin et al., 2004). Hence, the metamodel of SEA_ML can be considered as a PIMM, whilst the metamodels of JADEX (Kardas et al., 2009b), OWL and OWL-S are platform specific metamodels (PSMM) and model transformations between this PIMM and PSMMs pave the way for the MDD of the semantic web enabled MASs based on the OMG's well-known Model Driven Architecture (MDA) (OMG, 2003).

We chose JADEX (JADEX, 2003), as the target agent platform, since it is one of the well-known and frequently used agent platforms in agent research and development studies. Its open source Application Programming Interface (API) enables agent programmers to develop BDI agents. JADEX has an agent-oriented reasoning engine for coding rational agents with Extensible Markup Language (XML) and the Java programming language. The development of JADEX agents is based on a hybrid approach in which a declaration of static agent properties and the programming of executable agent plans take place. The declaration of static agent properties is given in files called Agent Definition Files (ADF). An ADF is written using XML and specifies the BDI model of the related agent. Moreover, agent plans are executable components which are given in Java program files. The JADEX reasoning engine starts the deliberation process by considering the goals requested by the agent. To this end, it adopts those goals stored in the database that contain all the adopted goals by the agent, called the agent's goal-base (Pokahr et al., 2007).

There is a base notation including three major terms regarding the JADEX architecture: beliefs, goals and plans. Beliefs are Java objects which represent the environmental facts that an agent have and are stored in a belief-base. The belief-base contains the facts that an agent possesses, in other words, it represents the knowledge of the world in which the agent is situated. Beliefs may change over the course of time within a dynamic environment, thus the belief-base needs to be updated in the long run.

The goals in JADEX resemble the desires discussed in (Rao and Georgeff, 1995) to some extent. However, the goals are a vital part of JADEX rather than the events in traditional BDI systems.

JADEX plans are Java classes which can be executed in order to achieve a goal of an agent. Plans have two parts: plan head and plan body. Furthermore, each agent has an ADF file, an XML-formatted file, to configure the agent's structure.

The metamodel of JADEX (Kardas et al., 2009b) reflecting the above discussed architecture is used as the target agent PSMM during model transformations from the SEA_ML instances in this study. On the other hand, OWL (W3C, 2004), W3C's standard language for the definition and development of ontologies is employed in the realization of SEA_ML's ontological concepts. As stated in its specification, OWL uses both Universal Resource Identifiers (URIs) for naming and the description for the Web provided by Resource Description Framework (RDF) (W3C, 1999), a standard model for data interchange on the Web, in order to add the ability of being distributed across many systems, scalability to Web requirements, compatibility with Web standards for accessibility and internationalization and finally openness and extensibility capabilities. OWL builds on RDF and RDF Schema and adds more vocabulary for describing the properties and classes like the relationship between classes, cardinality, equality, richer typing of properties, and the characteristics of properties and enumerated classes. We have adopted OMG's ODM (OMG, 2009) as the metamodel of OWL and used it during the transformations as another target PSMM.

It is also worth indicating that semantic web services modeled according to SEA_ML are transformed into OWL-S services for providing the implementation of these services. Based on OWL, OWL-S (Martin et al., 2004) introduces a top level service ontology in which three essential types of knowledge about a service can be stored. Service Profile tells what the service does and provides information for discovering a service. Service Model describes how the service can be used and includes the composition structure of the service. Finally, Service Grounding provides knowledge on interacting with the related service. The class Service in OWL-S provides an organizational point of reference for a declared Web service. One instance of Service exists for each distinctly published service. Main properties of the Service are named as *presents*, *described_by*, and *supports*. The classes *ServiceProfile*, *ServiceModel*, and *ServiceGrounding* are the respective ranges of these properties. Each instance of Service presents a *ServiceProfile* description, is described by a *ServiceModel* description, and supports a *ServiceGrounding* description (Martin et al., 2004). Hence, in our study, each SWS modeled in SEA_ML was transformed into an OWL-S Service class and proper Service Profile, Service Model and Service Grounding documents were generated for the related SWS.

After determining the entity mappings between SEA_ML and the above discussed target PSMMs, it is necessary to provide model transformation rules which are applied at runtime on SEA_ML instances to generate platform specific counterparts of these instances. For that purpose, transformation rules should be formally defined and written according to a model transformation language. To this end, many languages have been proposed (e.g. (Duddy et al., 2003; Kalnins et al., 2005; Agrawal et al., 2006; Jouault and Kurtev, 2006). In this study, we preferred to use ATL Transformation Language (ATL) to define the model transformations between SEA_ML and the target platforms (JADEX, OWL and OWL-S). ATL (Jouault et al., 2008) is one of the well-known model transformation languages which are specified as both metamodel and textual concrete syntax. An ATL transformation program is composed of rules that define how the source model elements are matched and navigated to create and initialize the elements of the target models. In addition, ATL can define an additional model querying facility which enables specifying the requests onto models (ATLAS Group, 2006). ATL also allows code factorization through the definition of ATL libraries. Finally, ATL has a transformation engine and an integrated development environment (IDE) that can be used as a plug-in on an Eclipse platform (The Eclipse Foundation, 2007a). These features of ATL caused us to prefer it as the implementation language for the transformations from SEA_ML.

ATL is composed of four fundamental elements. The first one is the header section which defines those attributes that are relative to the transformation module. The next element is the import section

which is optional and enables the importing of some existing ATL libraries. The third element is a set of helpers that can be viewed as the ATL equivalents to the Java methods. They make it possible for defining factorized ATL code that can be called from different points of an ATL transformation. The last element is a set of rules that defines the way target models are generated from source models. ATL uses Object Constraint Language (OCL) (OMG, 2012) expressions to control model transformations.

ATL is used for metamodels which have been developed based on the Eclipse Ecore meta-metamodel. Thus, SEA_ML's syntax has been defined in Ecore as discussed in Sections 2 and 3. In addition, we used Ecore representations of metamodels of the target platforms (JADEX BDI and OWL) and hence output models of the related MAS can be achieved after automatic execution of the defined transformation rules. In other words, we fed the M2M transformation based on ATL by employing the SEA_ML syntax (Ecore files of each viewpoint) as the input metamodel and the instance models conforming to the SEA_ML metamodel which are in the XMI format generated by our GMF-based tool. The value of this approach is twofold: First, the validation of all metamodels and models which conform to these metamodels is supported, since all the models obey EMF/Ecore rules. Second, our target metamodels already own the defined and executable semantics and application of transformations make use of these semantics for SEA_ML models, which constitutes the first step for real implementation of MASs modeled according to SEA_ML.

In order to provide some flavor of the transformations, entity mappings for some of the SEA_ML viewpoints and defined rules are discussed in the rest of this section. For instance, the mappings between the Agent Internal viewpoint of the SEA_ML metamodel and JADEX metamodel can be found in Table 5. Some meta-elements are used in two or more viewpoints but listed only in the related mapping table as one of those viewpoints. For example, although Belief is a meta-element of the Agent Internal viewpoint, it does not exist in Table 5 since it is considered in the Ontology viewpoint.

Table 5: Mappings between SEA_ML's Agent Internal viewpoint and JADEX Metamodels.

SEA_ML's Agent Internal viewpoint	JADEX
SemanticWebAgent	Agent
Behavior	Plan
Plan	Plan
Capabilities	Capability
Goal	AchieveGoal
Goal	QueryGoal
Goal	PerformGoal

Another group of transformation rules can be exemplified for the Agent-SWS Interaction viewpoint in which the related viewpoint of SEA_ML is treated as the source metamodel. In addition to JADEX, the metamodel of OWL-S is used as the target metamodel in this transformation. The entity mappings between these metamodels are shown in Table 6.

While the transformation rules are being defined, the source and target metamodels must be indicated in the ATL code, as shown in Listing 1. This information is also defined in the properties of the created ATL project on the Eclipse platform. As is shown in Listing 1, the "SWSInteraction.ecore" file is the input for the transformation rules, which is denoted by the “IN” keyword in line 4, while the "Jadex.ecore" file is the output for the transformation which is denoted by the “OUT” keyword in line 4.

Table 6: Mappings between SEA_ML Agent - SWS Interaction, JADEX and OWL-S metamodels.

SEA_ML's Agent - SWS Interaction viewpoint	JADEX	OWL-S
SemanticWebAgent	Agent	
SSMatchmakerAgent	Agent	
Plan	Plan	
SS_AgreementPlan	Plan	
SS_ExecutorPlan	Plan	
SS_FinderPlan	Plan	
SS_RegisterPlan	Plan	
SWS		Service
WebService		Service
Interface		ServiceProfile
Process		ServiceModel
Grounding		ServiceGrounding
Input		Input
Output		Output
Precondition		Condition
Effect		ResultVar

```
01  module SWSVP2Jadex;
02  -- @path SWSInteraction=/SWSVP2Jadex/SWSInteraction.ecore
03  -- @path Jadex=/SWSVP2Jadex/Jadex.ecore
04  create OUT: Jadex from IN: SWSInteraction;
```

Listing 1: Definition of metamodels for ATL rules.

An example of the transformation rule used to transform the SEA_ML's SemanticWebAgent to the JADEX's Agent is given in Listing 2, which includes the rule entitled “SemanticWebAgent2Agent”.

Giving meaningful names to the rules provides convenience and prevents confusion during encoding. As is shown in lines 2 and 3, source metamodel properties are indicated with the “from” keyword and target metamodel properties are indicated with the “to” keyword. This rule creates a JADEX Agent for every instance of the Semantic Web Agent (Line 3) and prepares the related properties and relationships of this instance. The Semantic Web Agent which is given in the source part of the rule (Line 2) can have more than one instance. While these instances are being identified, the helper rules are used to distinguish the instances and determine the relationships. These helper rules are also used in Listing 2 (Lines 2 and 7). The "description" and "property" attributes of the SemanticWebAgent are transformed into their JADEX Agent counterparts in lines 5 and 6 respectively.

```
01  rule SemanticWebAgent2Agent{
02     from swagent : SWSInteraction!SemanticWebAgent ( swagent.part1PatternforSWA )
03     to jagent : Jadex!Agent(
04         name <- swagent.setName(),
05         description <- swagent.description,
06         propertyfile <- swagent.properties,
07         plans <- Sequence{swagent.finderPlan, swagent.agreementPlan, swagent.executorPlan}
08      )
09  }
```

Listing 2: Transformation from SEA_ML SemanticWebAgent to JADEX Agent.

The Helper rules compose the constraint parts of the main rules. Constraints are used for querying source models. Constraints are prepared by using Object Constraint Language (OCL) (Warmer and Kleppe, 2003; OMG, 2012) in ATL. Usage of the same helper rules and repetition of the constraints may be required for the same target model or a different target model. Separation of these helper rules from the main rules supplies the usage of the same helper rules in different transformations.

Three types of helper rules were used in this study. The first one controls the empty strings. For example, if a "name" is not given to an Agent, it sets "AGENT_NAME_IS_EMPTY" as the name of the output file. If there is a name, then the helper rule controls the first letter and changes it to a lower case, in case it is not.

The second type of helper rules are used to control the input model's convenience for the intended pattern. For example, a Goal has a relationship with Plan and Capabilities in the metamodel. By using this type of helpers, the relationship of the received element is compared with the Plan and Capabilities relationship. If they match, it is decided that the element is a Goal. The last type of helper rules are used to generate the relationships of target elements by considering the source and target metamodels. For instance, Goal uses Capabilities, Capabilities applies Plan and Plan dispatches Goal within the source metamodel. By using these types of helpers, these relationships are defined and related elements are transformed into a target metamodel with their related elements.

The usage of all types of helper rules is exemplified in Listing 2. In line 4, we can see that the "name" attribute of the Agent meta-entity is obtained after executing the *swagent's* "setName" helper rule. The "setName" helper rule returns a string which is the name of the related SemanticWebAgent instance. The helper rule "*part1PatternforSWA*" in line 2, is an example of the second type of helper rule. It is through this helper, that all the Semantic Web Agents are determined in the input model. The "*part1PatternforSWA*" helper rule is given in Listing 3.

```
01 helper context SWSInteraction!SemanticWebAgent def: part1PatternforSWA : Boolean =
02     if   not self.oclIsTypeOf(SWSInteraction!SSMatchmakerAgent)  and
03          not self.plays.oclIsTypeOf(SWSInteraction!RegistrationRole)  and
04          self.apply -> select (p | p.oclIsTypeOf(SWSInteraction!SS_FinderPlan))
05              ->forAll(p  | p.discovers.oclIsTypeOf(SWSInteraction!Interface))  and
06          self.apply->select(p | p.oclIsTypeOf(SWSInteraction!SS_AgreementPlan))
07              ->forAll(p  | p.negotiates.oclIsTypeOf(SWSInteraction!Interface))  and
08          self.apply->select(p | p.oclIsTypeOf(SWSInteraction!SS_ExecutorPlan))
09              ->forAll(p  | p.executes.contains=p.use.supports)
10     then true else  false
11     endif;
```

Listing 3: *part1PatternforSWA* helper rule.

The *part1PatternforSWA* helper rule determines whether there is an instance of the SemanticWebAgent in the input model which is controlled by the constraints in Lines 2 to 9. These constraints control the Semantic Web Agent's relationships with the SSMatchmakerAgent, the RegistrationRole, the SS_FinderPlan, the Interface, the SS_AgreementPlan, and the SS_ExecutorPlan. If the input model element satisfies all the conditions, the helper rule returns "true" to the main rule; then, the main rule fulfills the transformation.

The "finderPlan", "agreementPlan", and "executorPlan" helpers (Line 7 in Listing 2) are examples of the third type. The related elements are chosen and the patterns determined by using these helpers. For instance, "executorPlan" helper selects the related Semantic Web Agent, Process and Grounding instances and then the main rule transforms these elements to the target model. The "executorPlan" helper is given in Listing 4 as an example.

```
01 helper context SWSInteraction!SemanticWebAgent def: executorPlan:
02 Sequence(SWSInteraction!SS_ExecutorPlan)  =
03        self.applies ->select (exeplan | exeplan.oclIsTypeOf  (SWSInteraction!SS_ExecutorPlan)  and
04        exeplan.appliedBy  ->forAll(agnt | not agnt.oclIsTypeOf(SWSInteraction!SSMatchmakerAgent)))
05        ->select(pln | pln.executes.contains = pln.use.supports and pln.appliedBy
06        ->select(agt | agt.oclIsTypeOf(SWSInteraction!SSMatchmakerAgent))  ->forAll(agt | agt.applies
07        ->select(p | p.oclIsTypeOf(SWSInteraction!SS_FinderPlan))
08        ->forAll(p  | p.interacts_with.advertises ->exists(intfc | intfc=p.discovers))));
```

Listing 4: executorPlan helper rule.

The “executorPlan” helper rule enables finding of those SS_ExecutorPlan instances which belong to a Semantic Web Agent in the input model. Thus, the *SS_ExecutorPlan*s of the related Agent instances in the target model are determined and their relationships are prepared in the main rule that will perform the transformation. The OCL constraints between lines 3 and 8 determine whether the Plan instance is an SS_ExecutorPlan by considering its relationships with the other model elements. Line 3 determines whether the Plan instance is an SS_ExecutorPlan which belongs to a Semantic Web Agent or not. Those Plan instances, that satisfy the constraints, return to the main rule as a query result of the helper rule. Similar helpers are used to select the appropriate elements for SWS in pattern matching.

Listing 5 controls a SWS's relationships with its WebService, Interface, Process, and Grounding members. Lines from 2 to 5 are constraints for controlling a SWS's relationships with the related elements. If the input model element satisfies all the conditions, the helper rule returns the “true” value to the main rule then the main rule provides transformation from source model to target.

```
01 helper context SWSInteraction!SWS def: part1PatternforService : Boolean =
02   if self.is_composed_of.oclIsTypeOf(SWSInteraction!WebService) and
03       self.presentedBy.oclIsTypeOf(SWSInteraction!Interface) and
04       self.describes.oclIsTypeOf(SWSInteraction!Process) and
05       self.supportedBy.oclIsTypeOf(SWSInteraction!Grounding)
06   then true else false
07   endif;
```

Listing 5: *part1PatternforService* helper rule.

The Interface's relationships are defined using a similar rule. Thus, the related input, output, precondition, and resultVar elements in the target model are determined (see Table 6). Listing 6 shows the “*part1PatternforInterface*” helper rule.

Line 3 in Listing 6 determines the related Semantic Web Agent and its Roles. Interface has a relationship with the SS_RegisterPlan, and the SS_RegisterPlan has a relationship with the Semantic Web Agent. This Semantic Web Agent must play a Role and the Role must be in an interaction with SWS as presented by the Interface. All the related elements are detected in this manner. The other lines in Listing 6 help to find the related SWS, Input, Output, Precondition, and Effect elements.

```
01   helper context SWSInteraction!Interface def: part1PatternforInterface : Boolean =
02     if self.presents.oclIsTypeOf(SWSInteraction!SWS) and
03        self.advertisedBy.appliedBy->exists(agnt|agnt.plays = self.presents.interactedBy) and
04        self.contains.oclIsTypeOf(SWSInteraction!Input)and
05        self.includes.oclIsTypeOf(SWSInteraction!Output)and
06        self.embodies.oclIsTypeOf(SWSInteraction!Precondition)and
07        self.involves.oclIsTypeOf(SWSInteraction!Effect)
08     then true else false
09     endif;
```

Listing 6: *part1PatternforInterface* helper rule.

Lastly, to give an example of ATL transformations in the Agent Internal viewpoint, the “*part1PatternforBehavior*” helper rule is given in Listing 7.

```
01   helper context Agent!Behavior def: part1PatternforBehavior : Boolean =
02     if self.includedBy.oclIsTypeOf(Agent!Role) and self.executes.oclIsTypeOf(Agent!Plan)
03     then true else false
04     endif;
```

Listing 7: *part1PatternforBehavior* helper rule.

The “*part1PatternforBehavior*” helper rule controls whether there is an instance of Behavior in the input model. The constraints in line 2 of Listing 7 control its relationships with the Role and Plan elements. If the input model element satisfies all the conditions, the helper rule returns “true” value to the main rule then the main rule provides the transformation. The OWL transformations are not discussed here due to their similarity of the transformation rules.

The platform-specific instance models, created after application of the above discussed ATL rules, can be opened later and modified within graphical modeling environments. For instance, in order to increase the quality of the throughput of the development process's next step, which is automatic code generation from the models (discussed in Section 5), a developer may wish to edit the generated JADEX model of the MAS to be implemented. To do this, Ecore encoded model file of the MAS is opened in a graphical modeling tool (Kardas et al., 2009b) for the JADEX BDI agents, visually edited, and then saved again in Ecore format for use in the next step: code generation.

5. Model-to-Text Transformations for Code Generation

Following the production of platform specific models over model transformations, a series of model-to-text (M2T) transformations are applied on these models in order to generate executable software codes for the MAS being implemented. In order to support this kind of interpretation of SEA_ML models, in this study, M2T transformation rules are written in MOFScript (Oldevik et al., 2005). MOFScript is a language specifically designed for the transformation of models into text files and deals directly with metamodel descriptions (Ecore files) as input. Also, it provides a tool as an Eclipse plug-in (The Eclipse Foundation, 2005) in which MOFScript transformations can be written, parsed, checked, and directly executed from the Eclipse environment.

MOFScript supplies the definition of code generation rules without dependency on any metamodel and also supports the interpretation of these rules. Hence, we first prepare the MOFScript rules and apply these rules on platform specific MAS models at runtime for the generation of JADEX BDI agent codes, OWL ontology files, and OWL-S documents corresponding to each designed semantic web service. The generated codes for MAS can be directly executed within the JADEX environment. The remainder of this section discusses some examples of the M2T transformation rules provided in this study.

When considering the JADEX structure, each agent should have an ADF which is an XML formatted file and codes of the related agent's plans in Java language. An ADF describes the structure of an agent. In other words, ADF defines the agent's elements. It defines the capabilities of an agent including beliefs, goals and plans. The Belief set is composed of fact variables which will be used with the plans of the agents. ADF files use reference names for those elements to be referred to within the plans. Hence, we prepared the rules for the generation of both the ADFs and Java plan classes of each modeled agent.

A part of the MOFScript codes for creating ADF files is given in Listing 8. The name of the M2T file is "JadexDiagram2JadexADF". The JADEX Ecore path is given by the "in" keyword in parentheses in the first line. In line 3, the generateAgentFile() method is invoked for every "agent" keyword using *forEach*. Also, the operation called *objectsOfType* is used to retrieve the contained model objects. The generateAgentFile() method's definition starts in line 5. For each invocation of the generateAgentFile() method, an agent.xml file is created in line 6. For example, if an agent, Agent1, has a name attribute associated with it, a file named agent1.agent.xml will be created. The lines between 7 and 9 represent a declaration of those namespaces which will exist in the ADF file. The name, description, and property attributes of the Agents are declared in an ADF file using the codes between lines 10 and 12. Consequently, we obtain ADF files which are XML files, for each agent. The beliefs, plans, goals, and capabilities of the agents are represented in these files.

The code block given in Listing 9 represents goal definitions in a generated ADF file. In an ADF file, there are definitions of related types of goals inside the '<goals>' tags. Related goal translation methods are invoked for each type of goal.

```
01 texttransformation JadexDiagram2JadexADF (in jadex:"http://jadex/5.0"){
02    main(){
03        jadex.objectsOfType(jadex.Agent)->forEach(agent){  agent.generateAgentFile() }
04    }
05    jadex.Agent::generateAgentFile(){
06       file (self.name+".agent"+".xml")
07        <%<agent xmlns="http://jadex.sourceforge.net/jadex"
08       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:
09       schemaLocation="http://jadex.sourceforge.net/jadex http://jadex.sourceforge.net/jadex-2.0.xsd"  %>
10       'name= "' self.name"'
11       'description= "' self.description"'
12       'properties="' self.properties"'
13    }
14 }
```

Listing 8: An excerpt from MOFScript rules generating JADEX ADF files.

```
1 '<goals>'
2   jadex.objectsOfType(jadex.Achievegoal)->forEach(achieveGoal){  achieveGoal.translateAchieveGoal()  }
3   jadex.objectsOfType(jadex.Maintaingoal)->forEach(maintainGoal){maintainGoal.translateMaintainGoal()}
4   jadex.objectsOfType(jadex.Performgoal)->forEach(performGoal){performGoal.translatePerformGoal()  }
5   jadex.objectsOfType(jadex.Metagoal)->forEach(metaGoal){  metaGoal.translateMetaGoal()  }
6   jadex.objectsOfType(jadex.Querygoal)->forEach(queryGoal){  queryGoal.translateQueryGoal()  }
7 '</goals>'
```

Listing 9: An excerpt from MOFScript rules which invokes goal definition rules.

The code of translateAchieveGoal() method is given in Listing 10 as an example. All the attributes of "Goal" meta-entity are captured and added to the ADF file during the code generation phase by these rules. As is shown in line 2, the attributes of the goals are declared. Corresponding methods for *parameterSet* and deliberation are invoked for each keyword parameter. JADEX Plan files are generated with similar rules. As can be seen in Listing 11, the generatePlanFile() method is triggered (Line 3), for each "body" keyword. All the plan file names are created by including the ".java" extension. Predefined packages are indicated by the writePackage() method. The remaining part of the rule has a similar working principle to that given previously in Listing 8.

```
01   jadex.Achievegoal::translateAchieveGoal(){
02       '<achievegoal name=" 'self.name' "'  'recur="'self.recur' " ' 'description = " 'self.description'" ' ' ' ">'
03       jadex.objectsOfType(jadex.achieveGoalParameters)->forEach(parameter){
04           parameter.translateParameter()
05       }
06       jadex.objectsOfType(jadex.achieveGoalParameterSets)->
07           forEach(parameterSet){  parameterSet.translateParameterSet()  }
08       jadex.objectsOfType(jadex.deliberateAchieveGoal)->  forEach(deliberation){
09           deliberation.translateDeliberation()
10       }
11   }
```

Listing 10: Some of the MOFScript rules for producing agent goal definitions.

```
01   texttransformation JadexDiagram2JadexPlan (in jadex:"http://jadex/5.0"){
02      main () {
03          jadex.objectsOfType(jadex.Body)->forEach  (body) { body.generatePlanFile() }
04      }
05      jadex.Body::generatePlanFile  (){
06          file (self.class +".java")
07          jadex.objectsOfType (jadex.Agent)->forEach(agent){  agent.writePackage() }
08      }
09   }
```

Listing 11: An excerpt from the MOFScript rules that generates JADEX plan codes.

In addition to the creation of JADEX agents, another group of rules is defined and implemented for the automatic generation of ontology files. A Web Services Description Language (WSDL) file and four OWL files are generated for each semantic web service. As discussed in Section 4, each OWL-S semantic web service is represented with a "Service.owl" file. Likewise, the service profile, service process, and service grounding of this semantic web service are described in files called "profile.owl", "process.owl", and "grounding.owl" respectively. Finally, we also generate the corresponding WSDL file (or files considering composite services) for each modeled SWS since the execution of the related SWS by agents, in fact, lies beneath the invocation of real web services that have WSDL interfaces. As is shown in Listing 12, ontology files are created for each service. Also, for each "service" keyword, an OWL-S Service file, an OWL-S Profile file, an OWL-S Process file, an OWL-S Grounding file, and a WSDL file are created in lines 4 to 8.

Through the MOFScript codes given in Listing 13, a "service.owl" file is generated including references to the "profile", "process", and "grounding" files of the SWS in question.

```
01  texttransformation OWLSTransformation (in owls:"/MyTest/model/OWLS.ecore"){
02      main () {
03          owls.objectsOfType (owls.Service)->forEach(service)  {
04              service.createOWLSServicefile()
05              service.createOWLSProfilefile()
06              service.createOWLSProcessfile()
07              service.createOWLSGroundingfile()
08              service.createWSDLfile()
09          }
10      }
11  }
```

Listing 12: An excerpt from the MOFScript rules that generates OWL-S Files.

```
01  <service:Servicerdf:ID= "'owls.Service.name'">
02  <!-- Reference to the Profile -->
03  <service:presentsrdf:resource="&'owls.Service.name'_profile;#'owls.ServiceProfile.name"'/>
04  <!-- Reference to the Process Model -->
05  <service:describedByrdf:resource=""&"owls.Service.name'_process;#'owls.ServiceModel.name"'/>
06  <!-- Reference to the Grounding -->
07  <service:supportsrdf:resource=""&"owls.Service.name'_grounding;#'owls.ServiceGrounding.name"'/>
08  </service:Service>
```

Listing 13: Some of the MOFScript rules which define an OWL-S Service file.

Similar to OWL-S production, OWL Transformations are implemented to generate ontology files. Each generated OWL Ontology is represented in a ".owl" file. Some of the MOFScript rules for Ontology viewpoint are given in Listing 14 which generate OWL documents.

```
01  main (){
02      odm.objectsOfType (odm.OWLOntology)->forEach(owl)  { owl.generateOntologyFile() }
03  }
04  odm.OWLOntology::generateOntologyFile(){
05      file (self.name+".owl")
06       '\n* Ontology file generated at: '+date()+'  '+time()
07        self.owlStatement->forEach  (statement: odm.OWLStatement){   statement.name  }
08  }
```

Listing 14: Some of MOFScript rules that generates OWL Files.

Transformations for other viewpoints including Environment, Role, Plan, and Interaction are provided similarly. The codes generated based on M2T transformations for these viewpoints extend ADFs and plan files of agents.

6. Case Study: Development of an Agent-based Expert Finding System

The development of an agent-based expert finding system is discussed in this section in order to both evaluate and provide some flavor of the use of SEA_ML and the proposed MDD approach. Let us consider that there exist web services for supplying the expert needs of people. Suppose that software agents in an expert finding system work on the web by using service ontologies, find candidate services, and then try to make an agreement with those services on behalf of their human users by taking into consideration QoS metrics. Starting from a motivating example, we discuss the development of the system in the following subsections.

6.1 Motivation

As a motivating example, consider the following scenario in which a user (person) requests an expert on communication services. The user Ann wants to communicate with her cousin Lee. However, she has had no contact with her for a long time and does not have Lee's contact information including what kind of communication services Lee uses, and those services which could be used to contact her (e.g. Social network, e-mail, VoIP, and mobile phone number).

First of all, Ann has to find the right person anyway. When she chooses to search, a graphical user interface is created automatically by her agent so that she can provide some information regarding the request about finding this person. She chooses the concept of a family tree from the filtering criteria in the user interface and then she enters the name, surname and relationship to her. Ann was a bit surprised when she received the results. She saw that Lee Smith could not be found, but the semantic matcher returned a person Lee Burke as a possible match according to the family ontology. If Ann's agent search was based on a traditional text search, Ann would not find her cousin. However, the use of a semantic matcher that works on ontological representation of the family tree enabled Ann to find her cousin with another surname. Ann was surprised since she did not know her cousin had just got married. Nevertheless, the photo of her cousin definitely confirmed that the system returned the right person.

In a semantic environment, such a system works using ontology graphs. Semantic matchers traversed the family ontology to find Ann's cousin. Traversing on the graph and inference based on this traversal can be accomplished by the applications of certain algorithms that basically deal with information extraction. These approaches and algorithms can find the required approximate node on an ontology tree. In fact, they succeed in finding the nearest nodes to the desired value. Details of this ontology traversal and semantic matching operation are beyond the scope of this paper. However, interested readers may refer to (Sycara et al., 2003) and (Li and Horrocks, 2003) for the general idea of semantic matching.

After finding the right person, according to Ann's request for communicating with Lee, in the next graphical user interface she is asked by her agent to choose the way she wants to communicate. Then she prefers audio talking with her cousin. Her agent offers to contact Lee via a VoIP using GoogleTalk (GTalk) (Google Co., 2002). This selection is made upon the intelligence behind which checks both the user's input like free talking and the user's information which is gathered automatically like connection bandwidths, applications installed on mobile devices, and so on. In addition, Ann decides to send Lee a bunch of flowers, considering her recent marriage. So, Ann asks her own agent to find the appropriate service. To do this, Ann enters the required information such as the desired flower's name, color, amount, and cost range to limit the selections. The agent considers these parameters along with some other QoS parameters like the distance of the flower shop and Lee's home address (which is automatically extracted), and the cost of the delivery service (as a composite service) accordingly. The result includes selecting a flower shop and a delivery service company which are altogether within the cost range of Ann's request.

6.2 System Design

In our system, the required result about communication and shopping services is gathered by the interaction between semantic services and agents in a MAS. Ann's request is held by a SWA inside a SS_FinderPlan instance according to the SEA_ML's agent-SWS interaction viewpoint. The SS_FinderPlan instance basically finds the appropriate semantic web services which have already been registered with a SS_RegisterPlan and returns the list of these services to a SSMatchmakerAgent to advise Ann's agent about candidate services. The discovery of the semantic services by the SS_FinderPlan is made semantically by traversing the Service Ontology. A graphical representation of an example of the Service Ontology structure is given in Figure 11 with ontology classes and their subclass-superclass relationships.

Ann's agent (which is an instance of SEA_ML SWA meta-entity) uses both the input given by Ann via the system interface and the information which is gathered by the agent, while traversing the ontology graph. As a result for the communication part of the request, using ontologies, the agent decides that "talking" is a kind of communication which is online. The input requires a talk thus a chat or social network are not considered. According to the information, bandwidth is not enough for multimedia video talking. So, it goes on searching for suitable media in the VoIP services in regard to the sub-ontology, which is illustrated in Figure 12.

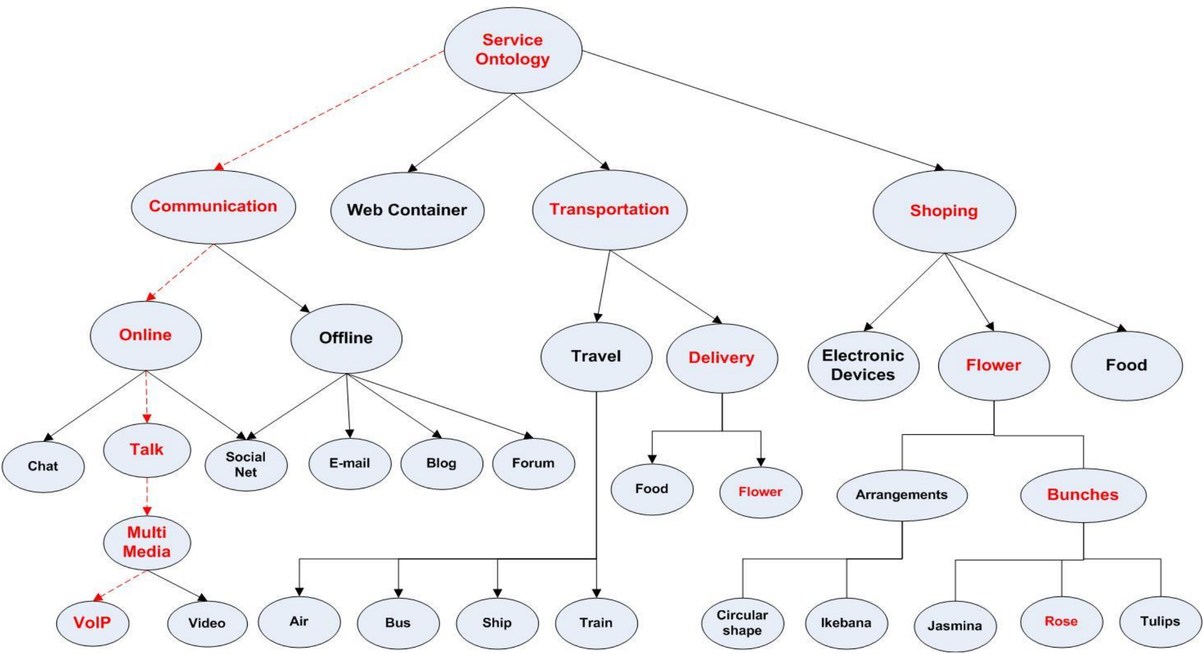

Figure 11: An example of the Service Ontology used in the expert finder system.

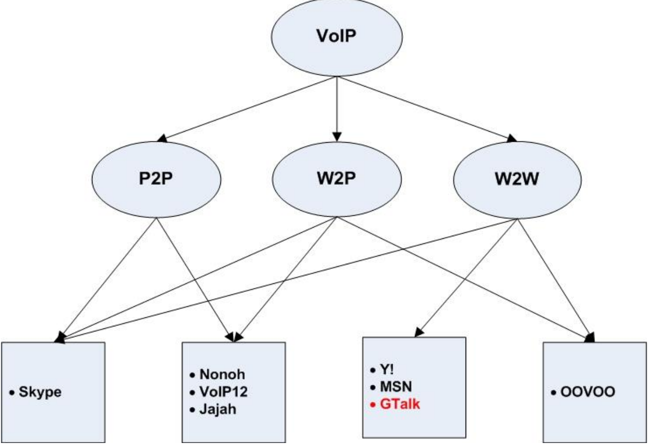

Figure 12: Sub-ontology for VoIP services.

Figure 12 is a sample VoIP service ontology. Ann's agent communicates with Lee's agent to query if Lee's phone has the required phone application. Since Ann and Lee's mobile phones have Internet connection, any kind of VoIP including phone to phone (P2P), web to phone (W2P), and web to web (W2W) is possible. However, when considering free calls, Nonoh (Nonoh, 2007) and Jajah (Jajah, 2006) are not included. In the same way, in regard to the applications installed on both mobile phones, Ann's agent's SS_FinderPlan provides the candidate services list containing GTalk and OOVOO (Oovoo, 2007) by interacting with the SSMatchmakerAgent. Then, the agent's SS_AgreementPlan chooses GTalk based on its sound quality.

In a similar manner, based on the human user request, the agent decides to buy the flowers from an e-flower shop, called "Beautiful Flowers", and sends them via a delivery company, called "Deliver Anywhere". The decision is made based on the QoS parameters which are both taken from Ann and extracted from the system automatically. For example, some of the flower shops are never considered in the result of the negotiation due to the type of flowers (see Figure 11), their colors (exact matching), and some others that are not selected due to the result of negotiation regarding the price. Furthermore, some of the delivery companies are omitted because of their delivery times, the delivering service was not available for small things like flowers, or as a result of negotiating on the cost of the delivery.

According to the scenario, we modeled the communication and shopping processes using SEA_ML. We consider all SWA agent instances in the MAS viewpoint aspect and then we evaluate each agent's internal structure within the Agent Internal viewpoint. After that, we model these agents' interactions with semantic web services and web service internal components.

6.3 System Modeling

The instance models with SEA_ML are achieved over MAS, Agent Internal, and Agent-SWS Interaction viewpoints. First, an overview of the system is modeled using the MAS viewpoint. The agents of this case study work in an organization named Communication organization. In addition, these agents, Ann's Agent ("MobileUser1"), Lee's Agent ("MobileUser2"), and the SSMatchmakerAgent ("ServiceMatcher") are in cooperation with each other. Basically, instance models are created under communication concepts. A screenshot of the MAS viewpoint instance model is shown in Figure 13. The "Communication" organization has its own role, called "Connector", and interacts with "Communication" environment.

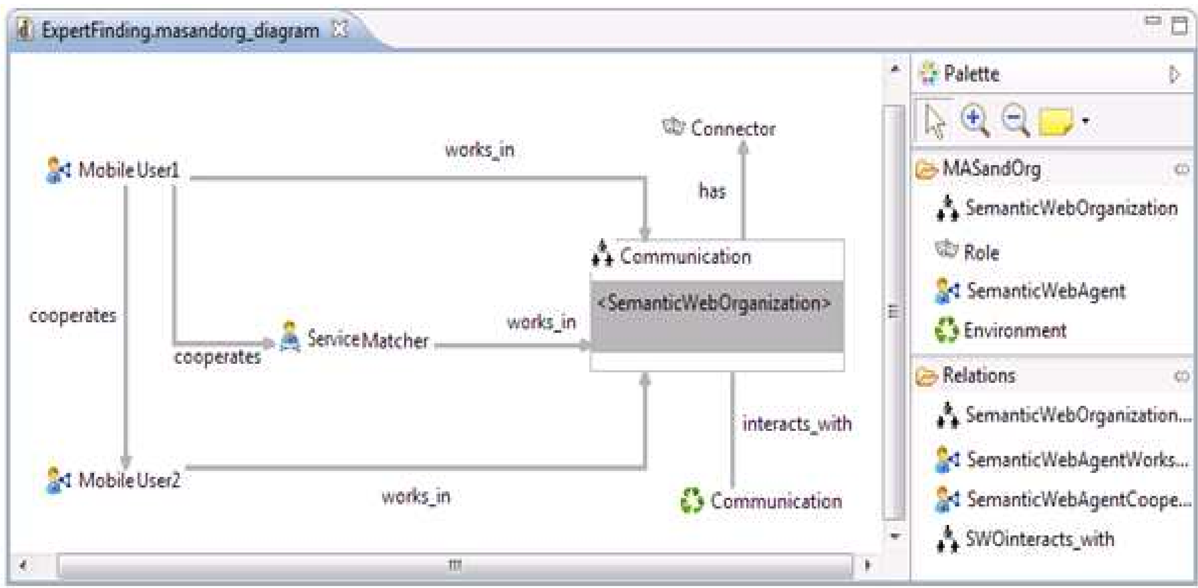

Figure 13: Graphical modeling for the MAS viewpoint of the Expert Finding system in the graphical syntax tool.

As an instance model of Agent Internal viewpoint, the screenshot in Figure 14 illustrates how Ann's agent's internal structure, as a SWA, can be modeled. It covers all of the required roles, behaviors and plans such as the "finding a person" plan. It also contains all the plan types which are covered in the Agent-SWS interaction viewpoint for discovering, negotiating and executing the candidate services (considering both the calling service and flower service). On the other hand, the agent capability with its goals and beliefs are also modeled. As belief instances for this scenario, "Family Knowledge" and "Shopping Knowledge" are modeled to be used with plans, like "finding a person" and "Lookup" respectively.

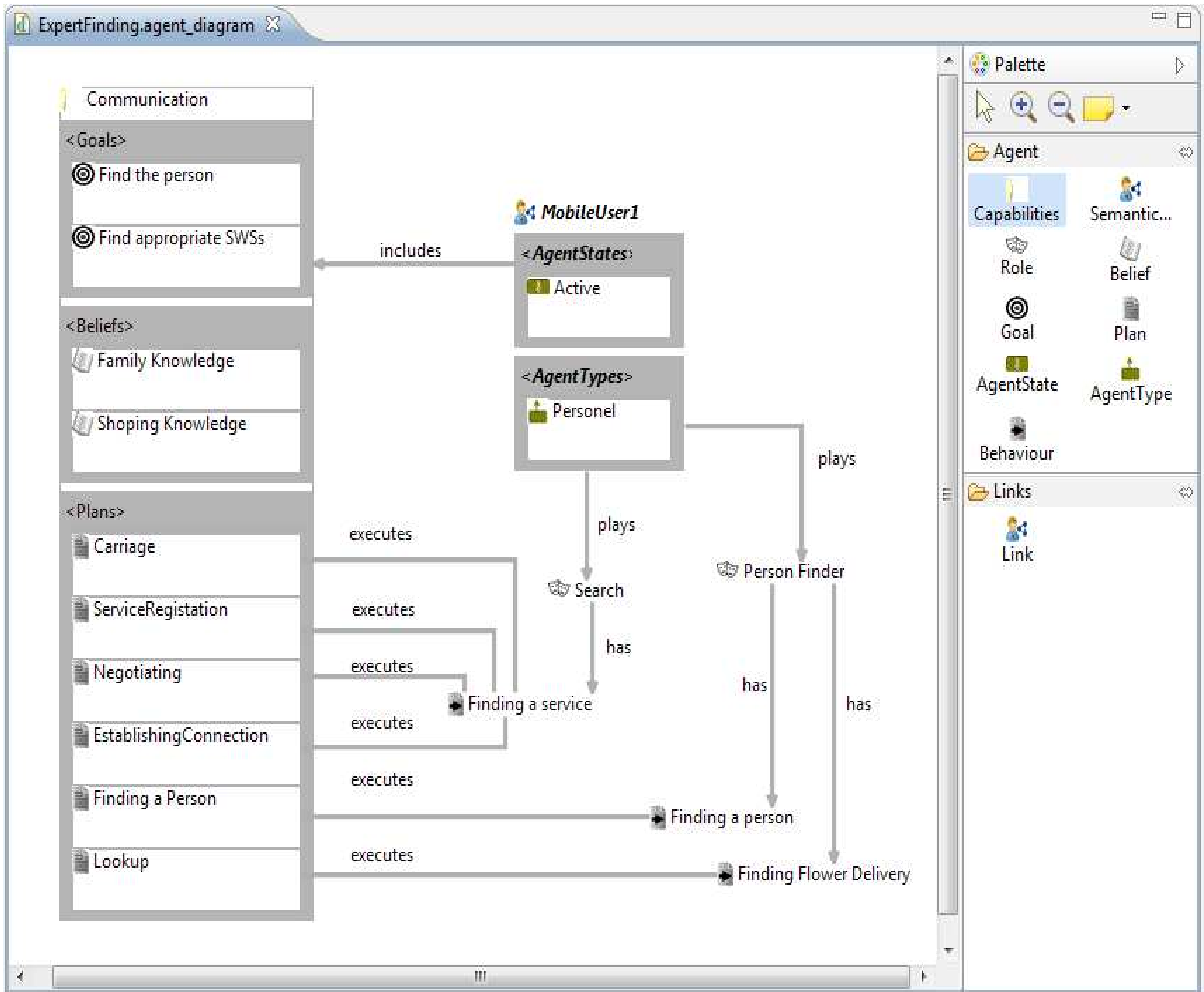

Figure 14: Graphical modeling for the Agent Internal viewpoint of a multiagent Expert Finding system in the graphical syntax tool of SEA_ML.

In Figure 15, a screenshot of the instance model for Agent-SWS interaction viewpoint is shown including semantic services and the required plan instances. Ann's agent, "MobileUser1", is modeled with proper plan instances to find, make the agreement with and execute the services which are the instances of the SS_FinderPlan, SS_AgreementPlan, SS_ExecutorPlan respectively. The services are also modeled with the interaction between the semantic web service's internal components (such as Process, Grounding, and Interface), and the SWA's plans.

So, when considering Ann's communication request, her agent plays the "Search" role and applies its "Lookup" plan to find an appropriate "Communication" interface of "Communication" SWS. This plan realizes the discovery via interacting with the "ServiceMatcher" which has registered the services by applying the "ServiceRegistration" plan. Next, the agent applies its "Negotiating" plan to negotiate

with the already discovered services. This negotiation is done through the "Communication" interface of the SWS. Finally, if the result of the negotiation is positive, the agent applies the "EstablishingConnection" plan to call the “Calling Web Service” of the SWS by executing its "Communicator" process and using its "ServiceProvider" grounding with which the service is realized.

In a similar manner, due to Ann’s request for flower shopping and delivery, her agent plays the "Search" role. It also applies the "Lookup" plan to find the "FlowerPresent" interface of the "FlowerDelivery" semantic web service, which is composed of the "E-Flower" and "Delivery" services. The "ServiceMatcher" and its "ServiceRegistration" plan help the "Lookup" plan in its goal. Then, the agent negotiates with the selected services’ interfaces and if the result is successful, the agent applies its specific execution plan, "Carriage", to call the "FlowerDelivery" SWS, by executing its "Shop&Deliver" process, and using its "FlowerProvider" grounding.

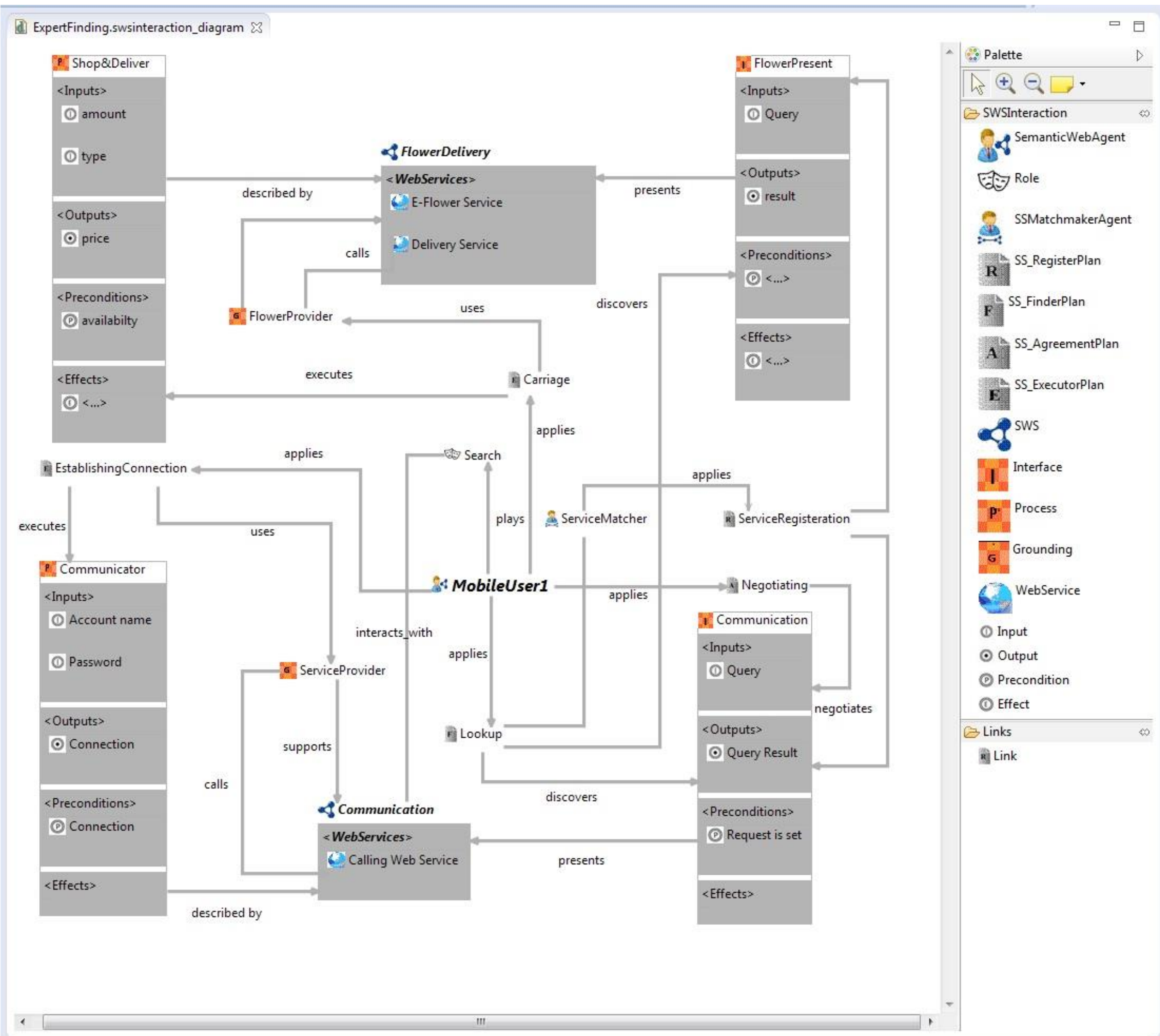

Figure 15: Graphical modeling for the Agent-SWS Interaction viewpoint of a multiagent Expert Finding system in the graphical syntax tool of SEA_ML.

6.4 System Development

Finally, the related code is generated after modeling the case-study using the SEA_ML's graphical tool. In order to do this, the previously discussed model-to-model and model-to-code transformations are automatically realized in order.

Artifacts gained as a result of the working on Agent Internal viewpoint and Agent-SWS Interaction viewpoints are discussed below in order to exemplify the generated code. As is mentioned in Section 4, the written ATL rules for SEA_ML are used for model-to-model transformations. As a result of model-to-model transformations in our case study, a graphical instance model is transformed into JADEX and OWL-S instance models.

For the Agent Internal viewpoint, a JADEX instance model for SEA_ML agents is achieved after the applications of the transformations. Following these transformations, the code for the related JADEX agents is generated by the application of written MOFScript rules, as explained in Section 5. However, the proposed approach also supports the developer's modification on the output JADEX model before generating the code. For this purpose, integration between the SEA_ML's tool and a platform-specific modeling tool (Kardas et al., 2009b) for JADEX agents is provided, as previously mentioned at the end of Section 4. Hence, the output JADEX model can be modified graphically (e.g. by inserting or deleting some elements or attributes), and is then given within the model-to-text transformation. The JADEX counterpart model for the Agent Internal viewpoint of our case study is illustrated in Figure 16.

Figure 16: Partial graphical model of the transformed Agent Internal viewpoint inside the platform-specific modeling tool (Kardas et al., 2009b) for JADEX agents.

Based on the mappings in Table 5, the elements in the Agent Internal metamodel are mapped to the elements in JADEX metamodel. Therefore, an automatic application of the ATL rules creates elements like “Find the person” and “Find appropriate SWSs” goals for the expert finder system. Similarly, the agent’s plans like “Finding a Person” plan are created as shown in Figure 16.

After ATL transformations, the output file is given to the MOFScript rules and as a result, an ADF file for agent, MobileUser1, and a plan file for each Plan element is generated. The generated ADF file is given in Listing 15.

In Listing 15, all of the meta-elements and their attributes correspond with the related tags of JADEX ADF. The “MobileUser1” agent’s capability, beliefs, and goals are defined in Lines 7, 11-14, and 18-22. All the attributes of the JADEX metamodel are not included in the SEA_ML metamodel. Therefore, lines 19 and 20 are default values of the corresponding attributes. In order to prevent repetition, only three of the Plan instances are given in Listing 15 (Lines 26 and 28). The “Carriage” plan, Line 26, is the execution plan of the “MobileUser1” agent for running the “FlowerDelivery” semantic web service. Similarly, the “EstablishingConnection” plan in Line 27 is the execution plan for running the “Communication” semantic web service. Finally, the “Lookup” plan, in Line 28, is the finder plan of the agent for finding both of the SWS services.

```
01 <agent xmlns="http://jadex.sourceforge.net/jadex"  xmlns:xsi="http://www.w3.org/2001/XMLSchema-
02      instance" xsi:schemaLocation="http://jadex.sourceforge.net/jadex-2.0.xsd"  name= "MobileUser1"
03      package= "" description= "" propertyfile="jadex.config.runtime"   abstract="false">
04      <imports> </imports>
05      <capabilities>
06          <capability>
07               name="Communication"   file="" description=""
08          </capability>
09      </capabilities>
10      <beliefs>
11          <belief name= "Family  Knowledge" class="" exported="" description="" updaterate=""
12               transient="" dynamic="">
13          <belief name= "Shopping Knowledge" class="" exported="" description="" updaterate=""
14               transient="" dynamic="">
15      </beliefs>
16      <goals>
17          <achievegoal
18               name="find  the person" recur="false" description=""
19               exclude="when_tried" exported="false"  posttoall="false" randomselection="false"
20               recalculate="true" recurdelay="0"  retry="true" retrydelay="0">
21               <creationcondition>  <!--  Write Conditions -->  </creationcondition>
22               …
23          </achievegoal>
24      </goals>
25      <plans>
26          <plan name="Carriage"  description="" exported="false"  priority="0">  </plan>
27          <plan name= "EstablishingConnection" description="" exported="false"  priority="0">  </plan>
28          <plan name="Lookup"  description="" exported="false"  priority="0">  </plan>
29          …
30      </plans>
31      …
32 </agent>
```

Listing 15: An excerpt from the generated ADF file for the Agent Internal viewpoint of MobileUser1 in “Expert Finding System” case study.

Two types of ATL rules are used for the Agent-SWS Interaction viewpoint. One for the multiagent part of the system (JADEX agents) which adds required SWS interaction code into the previously generated ADFs of the JADEX agents and another for the Semantic Web Service part of the system (OWL-S metamodel), which helps to generate OWL-S documents. Listing 16 shows the instance target model generated by ATL transformations for this viewpoint of the case study.

The model elements are presented in XMI format. For example, the “Communication” and “FlowerDelivery” semantic web services are defined in Lines 5 and 6. Process, Interface and Grounding are defined for “Communication” SWS in Lines 8, 10, and 12; and for “FlowerDelivery” SWS in Lines 9, 11, and 13.

With applying the ATL rules, two ADF files and six plan files are generated when considering the “MobileUser1” and “ServiceMatcher” agents. Also, a total of eight OWL-S files, four for each SWS (Service, Service Process, Service Profile, and Service Grounding) and two WSDL files, one for each SWS, are generated. In this case study, 504 lines of code (LOC) is generated by applying approximately 50 ATL M2M transformation rules and 40 MofScript M2T transformation template functions.

The ADF and plan files generated from the Agent-SWS interaction viewpoint are structurally similar to those generated from the Agent Internal viewpoint for “MobileUser1” agent. Part of ADF file for “MobileUser1” agent is shown in Listing 15.

```
01 <?xml  version="1.0" encoding="ISO-8859-1"?>
02 <xmi:XMI  xmi:version="2.0"  xmlns:xmi="http://www.omg.org/XMI"
03      xmlns:jfb.examples.gmf.OWLS="/MyTest/model/OWLS.ecore">
04      <jfb.examples.gmf.OWLS:Service>
05           <name> Communication  </name>
06           <name> FlowerDelivery  </name>
07      </jfb.examples.gmf.OWLS:Service>
08      <jfb.examples.gmf.OWLS:ServiceModel  name="Communicator"/>
09      <jfb.examples.gmf.OWLS:ServiceModel  name="Shop&Deliver"/>
10      <jfb.examples.gmf.OWLS:ServiceProfile   name="Communication"/>
11      <jfb.examples.gmf.OWLS:ServiceProfile   name="FlowerPresent"/>
12      <jfb.examples.gmf.OWLS:ServiceGrounding  name="ServiceProvider"/>
13      <jfb.examples.gmf.OWLS:ServiceGrounding  name="FlowerProvider"/>
14      <jfb.examples.gmf.OWLS:Input  name="Query"/>
15      <jfb.examples.gmf.OWLS:Input  name="Account name"/>
16      <jfb.examples.gmf.OWLS:Input  name="Password"/>
17      <jfb.examples.gmf.OWLS:Input  name="amount"/>
18      <jfb.examples.gmf.OWLS:Input  name="type"/>
19      <jfb.examples.gmf.OWLS:Output  name="Query Result"/>
20      <jfb.examples.gmf.OWLS:Output  name="Connection"/>
21      <jfb.examples.gmf.OWLS:Condition  name="Request is set"/>
22      <jfb.examples.gmf.OWLS:Condition  name="availability"/>
23      …
24      <jadex:JadexPlatform>
25            <plan name="ServiceRegistration"/>
26            <plan name="Lookup"/>
27            <plan name="Negotiating"/>
28            <plan name="EstablishingConnection"/>
29            <plan name="Carriage"/>
30            <agent name="MobileUser1"/>
31            <agent name="ServiceMatcher"/>
32      </jadex:JadexPlatform>
33 </xmi:XMI>
```

Listing 16: Generated instance model for the Agent-SWS-Interaction viewpoint.

```
01 <?xml  version="1.0"?>
02 <rdf:RDF  xmlns:rdf=  "&rdf;#" xmlns:rdfs="&rdfs;#"  xmlns:owl  = "&owl;#" xmlns:service=  "&service;#"
03     xmlns:ServiceProfile=&profile;#  xmlns:ServiceProfile=&process;#xmlns:ServiceGrounding=&
04     grounding;# xmlns="&DEFAULT;#"  xml:base="&DEFAULT;">
05     <owl:Ontology  rdf:about="">
06          <owl:versionInfo>
07                  $Id:Service.owl  generated at: 20/03/2013  11:8:23  am
08          </owl:versionInfo>
09          <rdfs:comment>
10                  This ontology represents the OWL-S  service description for the Service example.
11          </rdfs:comment>
12          …
13          <owl:imports  rdf:resource= "&Communication_profile;"  />
14          <owl:imports  rdf:resource= "&Communicator_process;"  />
15          <owl:imports  rdf:resource="&ServiceProvider_grounding;" />
16     </owl:Ontology>
17     <service:Service  rdf:ID=  "Service">
18            <service:presents rdf:resource="&Communication_profile;#  ServiceProfile'/>
19            <service:describedBy  rdf:resource=&Communicator_process;#ServiceModel'/>
20            <service:supports rdf:resource=&ServiceProvider_grounding;#ServiceGrounding'/>
21      </service:Service>
22      <!--  Inverse links  -->
23      <profile:Profile  rdf:about=&Communication_profile;#ServiceProfile/>
24     <service:presentedBy rdf:resource=#Service/>
25     …
26 </rdf:RDF>
```

Listing 17: An excerpt from a generated OWL-S Service file (“Service.owl”) for Communication SWS.

An excerpt from an OWL-S Service file is given as an example of generated SWS files in Listing 17. The "Service.owl" file includes references to Service Grounding, Service Model, and Service Profile files, which are generated for the "Communication" SWS. Lines 21, 22 and 23 of Listing 17 cover these references for the Profile, the Process Model, and the Grounding of the SWS respectively.

7. Related Work

Since the model-driven development of MASs is one of the major research topics in Agent-oriented Software Engineering (AOSE), researchers have proposed various agent metamodels and modeling languages which can guide the agent programmers during the development. Many proposed DSLs and DSMLs for MASs originate from or use the artifacts of these metamodels and modeling language studies. Hence, we prefer to group the related work into two subsections: Agent metamodel and agent general modeling language studies are discussed in the first subsection, while the remaining MAS DSL and DSML approaches are discussed in the second.

7.1 Agent and MAS Metamodels and Modeling Languages

AALAADIN (Ferber and Gutknecht, 1998) appeared as the first general-purpose metamodel for MASs. This metamodel represented a MAS structure with just three main concepts (agent, group and role) and their relationships. On the other hand, some AOSE researchers have preferred to define agent metamodels which are specific for their MAS development methodologies. For instance, Bernon et al. (Bernon et al., 2005) gave metamodels for the ADELFE (Bernon et al., 2003), Gaia (Zambonelli et al., 2003) and PASSI (Cossentino and Potts, 2002) MAS development methodologies and introduced a unified metamodel composed by merging the most significant contributions of these methodologies. The study also included a comparison of these three metamodels by considering their support on agent structure, agent interactions, the agent society, and the organizational structures and implementations of agents. A similar study (Molesini et al., 2005) introduced a metamodel for SODA (Omicini, 2000) agent development methodology. This study aimed to model the interaction and social aspects of the SODA and define a metamodel by considering these aspects. Apparently, those metamodels presented the formal representations for the concepts and associations between the concepts of the related methodologies. This was important and very helpful especially when we consider clear analysis and appropriate extension of the methodologies. Besides, Pavon et al. (Pavon et al., 2006) stated that these metamodels were also used within the agent community as tools that could help to compare different methodologies. However, it should also be noted that they were not suitable for general MAS modeling since the proposed models were specific to related methodologies and mostly provided mappings to very specific implementation frameworks or even sometimes did not deal with implementation at all (Pavon et al., 2006).

During recent years, the Technical Committee of IEEE FIPA (FIPA Modeling TC, 2003) and OMG made an effort on MAS metamodeling and developed a general agent metamodel called the Agent Class Superstructure Metamodel (ACSM) (Odell et al., 2005) in order to express relationships between agents, agent roles, and agent groups in a MAS. The specification of ACSM was both based on and extends the Unified Modeling Language (UML) 2.0 superstructure. However, with definitions of just 8 basic meta-entities and their relationships, ACSM was too abstract and needed extensions for the exact modeling of a MAS especially when we consider the internal agent behaviors and agent communications. For instance, in order to model the interactions between the agents and the semantic web services, Kardas et al. (Kardas et al., 2009a) proposed an agent metamodel which extended the ACSM and used that metamodel within their model-driven MAS development methodology. Moreover, (Bauer and Odell, 2005) introduced the specification of agent-based systems using UML 2.0. In fact, they discussed which aspects of a MAS could be considered as the Computation Independent Model (CIM) and the Platform Independent Model (PIM). Another significant MAS metamodel was introduced in (Hahn et al., 2009) which collects agent modeling concepts in seven MAS viewpoints called Multiagent, Agent, Behavioral, Organization, Role, Interaction, and

Environment. Hahn et al. employed this agent metamodel as a PIMM in the development of agent systems and achieved MAS executables in the same manner with (Kardas et al., 2009a).

The Agents & Artifacts (A&A) metamodel introduced in (Omicini et al., 2008) considered the notion of artifacts for agents. In the A&A metamodel, agents are modeled as proactive entities for the systems' goals and tasks while the artifacts represented the reactive entities providing the services and functions and, hence, constituted the environment for MAS. The FAML metamodel, introduced in (Beydoun et al., 2009), was in fact a synthesis of various existing metamodels for agent systems. Design time and runtime concepts for MASs were given and validation of these concepts was provided by their use at various MAS development methodologies.

The architecture of MAS software should inevitably be based on a proper modeling of the related MAS. Therefore, AOSE researchers have made great efforts on developing MAS modeling languages in addition to MAS metamodeling studies. Since UML (OMG, 2000) is the widely-accepted software modeling language, important MAS modeling studies are mostly based on the UML or provide agent-based extensions to UML. For instance, Depke et al. introduced an agent-oriented modeling technique (Depke et al., 2001) based on the UML notation. Graph transformation was used both on the level of modeling for capturing agent specific aspects and as the underlying formal semantics of the approach. In order to capture cooperation among several agents, Depke et al. employed graph transformation rules in the requirement specification and analysis.

Agent UML (AUML) (Bauer et al., 2001) is perhaps the more well-known modeling language in the agent community. AUML presents new agent-based extensions to package and template structures and sequence, interaction, activity and class diagrams of UML. Model semantics are represented with a metamodel and agent protocols are defined with a three-layered notation: The first layer defines the overall protocol with UML package and template structures. The second layer defines the interactions between agents by extending sequence, collaboration, interaction, and the activity diagrams and statecharts of UML. Finally, the third layer defines an agent's internal processing by using UML activity diagrams and statecharts again. Although AUML became slightly popular over the last decade in the MAS research community, experiences have shown that its UML extensions also bring some deficiencies. AUML relies too much on UML which is proposed for object-oriented system specification. This dependency causes the inability of AUML to be abstract enough from object considerations. As also stated in (Huget, 2005), AUML's visual notation is incomplete and does not provide a textual notation to exchange with other developers. Finally, AUML semantics is semi-formal and again based on the UML.

The Agent Modeling Language (AML) (Cervenka et al., 2005) is another general modeling language for MASs. Based on the UML 2.0 superstructure, AML provides a visual modeling of agent systems. AML is newer than AUML and has a more complete set of notations. However, utilizing all the different symbols of AML's notation is too complicated and difficult (Huget, 2005). Also, AML does not have a textual notation and it lacks semi-formal semantics.

7.2 DSLs and DSMLs for MASs

Although the above mentioned studies contribute to AOSE research within the perspectives of agent modeling and model-driven MAS development, studies on DSLs/DSMLs for agents have recently emerged and these very few studies are in their preliminary states. For instance, a DSL called Agent-DSL was introduced by (Kulesza et al., 2005). The Agent-DSL is used to specify these agency properties that an agent needs to accomplish its tasks. However, the proposed DSL is presented only with its metamodel and provides just a visual modeling of the agent systems according to agent features, like knowledge, interaction, adaptation, autonomy and collaboration. Likewise, in (Rougemaille et al., 2007), the authors introduced two dedicated modeling languages and call these languages DSMLs. These languages are described by metamodels which can be seen as representations of the main concepts and relationships identified for each of the particular domains again introduced in (Rougemaille et al., 2007). However, the study obviously included only the abstract syntax of the related DSMLs and does not give the concrete syntax or semantics of the

DSMLs. In fact, the study only defined generic agent metamodels for the model-driven development of MASs.

Originating from a well-formalized syntax and semantics, Ciobanu and Juravle defined and implemented a language for mobile agents in (Ciobanu and Juravle, 2012). Similar to our methodology but using different modeling and implementation technologies, a high-level DSL for mobile agents was achieved. Ciobanu and Juravle generated a text editor with auto-completion and error signaling features and they presented a way of code generation for agent systems starting from their textual description. The introduced DSL considered the mobile agents domain which completely differed from the specific domain of SEA_ML.

Hahn (Hahn, 2008) introduced a DSML for MAS. The abstract syntax of the DSML was derived from a platform independent metamodel which was structured into several aspects each focusing on a specific viewpoint of a MAS. This approach resembled to our study. In order to provide a concrete syntax, the appropriate graphical notations for the concepts and relations were defined (Warwas and Hahn, 2008). The semantics of the language were given in (Hahn and Fischer, 2009). This study was noteworthy because it seemed to be the first complete DSML for agents with all of its specifications. However, it supported neither the agents on the Semantic Web nor the interaction of Semantic Web enabled agents with other environment members like semantic web services. Our study contributes to the aforementioned efforts by also specializing in the Semantic Web support of MASs.

In (Hahn et al., 2008), the authors introduced their approach on integrating agents with semantic web services. In addition to the MAS metamodel described in (Hahn, 2008), a new platform independent metamodel was proposed for semantic web services. A relationship between these two metamodels was established in such a way that the MAS metamodel was extended with new meta-entities in order to support semantic web services interoperability, and it also inherited some meta-entities from the metamodel proposed for semantic web services. Instead of using two separate metamodels, SEA_ML has a built-in support for the modeling of agent and semantic web services' interactions by including a special viewpoint. Moreover, the semantic internal components of agents, like an agent's knowledgebase, could also be modeled in SEA_ML.

Another DSML was provided for MASs in (Gascuena et al., 2012). The abstract syntax was presented using the Meta-object Facility (MOF) (OMG, 2002), the concrete syntax and its tool was provided with GMF (The Eclipse Foundation, 2006), and finally the code generation for the JACK agent platform (AOS, 2001) was realized with model transformations using JET (The Eclipse Foundation, 2007b). However, the developed modeling language was not generic since it was based on only the metamodel of one of the specific MAS methodologies called Prometheus (Padgham and Winikoff, 2004). A similar study was done recently in (Fuentes-Fernandez et al., 2010) which proposes a technique for the definition of agent oriented engineering process models and can be used to define processes for creating both hardware and software agents. This study also offers a related MDD tool using Software & System Process Metamodel (SPEM) (OMG, 2008) and based on INGENIAS methodology (Pavon et al., 2005) for MAS development. Nevertheless, neither (Gascuena et al., 2012) nor (Fuentes-Fernandez et al., 2010) covered software agents in the Semantic Web.

By considering our previous studies, in (Kardas et al., 2010), we have shown how that domain-specific engineering can provide an easy and rapid construction of a Semantic Web enabled MASs. Ideas for abstract syntax, concrete syntax, and formal semantics have been discussed. Furthermore, a metamodel, which in fact constitutes the preliminary version of the abstract syntax of SEA_ML, was introduced in (Challenger et al., 2011). Also, a graphical tool, that can be used during both the modeling of Semantic Web enabled MASs and the syntactic checking of designed models, was presented in (Getir et al., 2011). Based on these building blocks, in this paper, we first discussed the complete infrastructure of SEA_ML including its syntax and semantics definitions, and showed how the language and its tools can be used during the development of real MASs.

8. Conclusion

This paper presented a DSML called SEA_ML for MAS[1]. The specification, implementation, and use of the proposed DSML were all discussed. In addition to the well-known aspects of a MAS, use of SEA_ML also provides quick and easy design and implementation of the interaction between autonomous agents and semantic web services inside the Semantic Web environment. The introduced metamodel of SEA_ML was specified in several viewpoints which may help agent developers in simplifying understanding of the problem space for various MAS realizations. Besides, the definition of such a metamodel led to the production of a graphical concrete syntax that can be employed for the platform independent modeling of Semantic Web enabled MASs without consideration of any deployment constraints or issues.

The model-centric MAS development approach presented herein also takes into account the exact MASs implementations such that the executables (required software codes) can be automatically achieved by using SEA_ML's operational semantics which are based on a series of model-to-model and model-to-text transformations. Hence, concrete realizations of the designed agents and semantic web services can be easily and rapidly generated as JADEX BDI agents (Pokahr et al., 2005) and OWL-S (Martin et al., 2004) instances, respectively. Also, with the case study discussed in this paper, we experienced all the above-mentioned features of SEA_ML including the use of its Eclipse-based tool. Agent developers may use this graphical tool of SEA_ML during all steps of the proposed MDD methodology; from platform independent modeling to automatic code generation for MASs working on the Semantic Web.

SEA_ML has advantages and disadvantages which can be, to some extent, attributed to many DSMLs. First, the end-user productivity has been greatly improved (from the example presented in Section 6 (Figures 15 – 17) 504 lines of code written in different files have been automatically generated). Second, as was shown in (Kosar et al., 2012) DSMLs improve the understandability of models/programs. As a consequence, readability, reasoning, and maintenance of models/programs are enhanced. Third, SEA_ML incorporate domain knowledge of MASs and hence enables reuse of this knowledge. As a consequence, mapping from the SEA_ML to the JADEX platform can be considered as efficient as that written by a domain expert. However, DSMLs are not a panacea for all software engineering problems. DSMLs' main disadvantage is their high development costs, and in this respect, SEA_ML is not an exception. Despite using appropriate tools and techniques (Mernik et al., 2005) SEA_ML development was accomplished over 18 month period. Please note, that its domain is complex and SEA_ML cannot be regarded anymore as a little language (see Sections 2-5).

In order to convey our experience considering both the DSML developers and the SEA_ML users, it is worth discussing some of the challenges and difficulties which we faced during language implementation and tool generation for SEA_ML. Regarding the experiences of the developers, the tools have some shortcomings. Eclipse GMF has some problems while generating the source code. GMF is challenging in some cases, for example, when one tries to have an instance of a super-class element and an instance of its sub-class element, simultaneously. This is not possible directly. Inheritance can be handled in two ways in GMF. One way is to create a separate eClass in the Ecore and assign the properties of super-type to that class. Then super-type and subtype will be extended from that eClass. The other approach is assigning all of the properties of the super-class to the subclass manually.

Regarding the users of the SEA_ML, there are some limitations. First, the ontologies should be provided with other tools such as Protégé (Protégé, 2004) or Jena (Jena, 2011). The next issue is that there are several well-known different SWS technologies, like OWL-S, WSMO (WSMO, 2005), and WSDL-S (WSDL-S, 2005). Therefore, having the important aspects of all of these technologies in

---

[1] Complete SEA_ML tools, including Ecore-encoded SEA_ML metamodel in 8 viewpoints, GMF-based editor, M2M transformation rules written in ATL and M2T transformation rules written in MofScript, along with the instance model and codes of the case study, and instructions for running them are all available as a bundle at: http://mas.ube.ege.edu.tr/downloads/sea_ml_bundle.zip. The bundle also includes more case studies demonstrating SEA_ML use in different application domains.

mind for a single metamodel is not easy and one technology will be stronger than the others. The current SWS notion, related entities, and M2M transformation mechanism in SEA_ML mostly support easy and rapid implementation of the semantic web services according to OWL-S due to its popularity and wide usage. However, since SEA_ML metamodel is platform independent, it is straightforward to define new M2M (and then M2T) transformations for the semantic web services modeled according to SEA_ML into other SWS technologies such as WSMO.

Our next work will consider the enrichment of SEA_ML's platform-specific support; such that agent systems designed according to SEA_ML specifications also could be implemented and executed in various agent platforms (e.g. JADE (Bellifemine et al., 2001) or JACK (AOS, 2001)). In order to provide this, we would first need to achieve a metamodel of those agent platforms and then build-up model-to-model transformations from SEA_ML's syntax to those platforms' models and, finally, define the model-to-text transformations in order to gather MAS executables for those platforms. The methodology to be applied for this future work would be similar to the one described in this paper for the JADEX platform.

Acknowledgment

This study was funded as a bilateral project by the Scientific and Technological Research Council of Turkey (TUBITAK) under grant 109E125 and the Slovenian Research Agency (ARRS) under grant BI-TR/10-12-004. The authors also wishes to thank the anonymous reviewers for their accurate comments on the previous versions of the paper. The authors have been able to improve both their work and the paper significantly by taking these anonymous reviewers' critical comments into account.

References

(Agrawal et al., 2006) Agrawal, A., Karsai, G., Neema, S., Shi, F., and Vizhanyo, A., 2006.The design of a language for model transformation. Software and Systems Modeling, 5(3), pp. 261-288.

(AOS, 2001) AOS, Agent Oriented Software Pty., Ltd., 2001. JACK Environment. available at: http://www.aosgrp.com/products/jack/ (last access: March 2013).

(ATLAS Group, 2006) ATLAS Group, LINA & INRIA, 2006. ATL User Manual. available at: http://www.eclipse.org/m2m/atl/doc/ATL_User_Manual[v0.7].pdf (last access: March 2013).

(Bădică et al., 2011) Bădică, C., Budimac, Z., Burkhard, H., and Ivanović, M., 2011. Software Agents: languages, tools, platforms. Computer Science and Information Systems, 8(2), pp. 255-298.

(Bădică et al., 2012) Bădică, C., Ilie, S., Kamermans, M., Pavlin, G., Penders, A., and Scafes, M., 2012. Multi-Agent Systems, Ontologies and Negotiation for Dynamic Service Composition in Multi-Organizational Environmental Management. Software Agents, Agent Systems and Their Applications, NATO Science for Peace and Security Series - D: Information and Communication Security, 32(12), pp. 286-306.

(Bauer et al., 2001) Bauer, B., Muller, J. P., and Odell, J., 2001. Agent UML: A formalism for specifying multiagent software systems. International Journal of Software Engineering and Knowledge Engineering, 11(3), pp. 207-230.

(Bauer and Odell, 2005) Bauer, B., Odell, J., 2005. UML 2.0 and agents: how to build agent-based systems with the new UML standard. Engineering Applications of Artificial Intelligence, 18, pp. 141–157.

(Berners-Lee et al., 2001) Berners-Lee, T., Hendler, J., and Lassila, O., 2001. The Semantic Web. Scientific American, 284(5), pp. 34-43.

(Bernon et al., 2003) Bernon, C., Gleizes, M-P., Peyruqueou, S., and Picard, G., 2003. ADELFE: A methodology for adaptive multi-agent systems engineering. Lecture Notes in Artificial Intelligence, 2577, pp. 70-81.

(Bernon et al., 2005) Bernon, C., Cossentino, M., Gleizes, M-P., Turci, P., and Zambonelli, F., 2005. A Study of some Multi-Agent Meta-Models. Lecture Notes in Computer Science, 3382, pp. 62-77.

(Bellifemine et al., 2001) Bellifemine, F, Poggi, A, and Rimassa, G., 2001. Developing Multi-Agent Systems with a FIPA-compliant Agent Framework. Software: Practice and Experience, 31(2), pp. 103-128.

(Beydoun et al., 2009) Beydoun, G., Low, G. C., Henderson-Sellers, B., Mouratidis, H., Gomez-Sanz, J. J., Pavon, J., and Gonzalez-Perez, C., 2009. FAML: A Generic Metamodel for MAS Development. IEEE Transactions on Software Engineering, 35(6), pp. 841-863.

(Bryant et al., 2011) Bryant B. R., Gray, J., Mernik, M., Clarke, P. J., France, R. B., Karsai, G., 2011. Challenges and Directions in Formalizing the Semantics of Modeling Languages. Computer Science and Information Systems, 8(2), pp. 225-253.

(Cervenka et al., 2005) Cervenka, R., Trencansky, I., Calisti, M., and Greenwood, D., 2005. AML: Agent Modeling Language–Toward Industry-Grade Agent-Based Modeling. Lecture Notes in Computer Science, 3382, pp. 31-46.
(Challenger et al., 2011) Challenger, M., Getir, S., Demirkol, S., and Kardas, G., 2011. A Domain Specific Metamodel for Semantic Web enabled Multi-agent Systems. Lecture Notes in Business Information Processing, 83, pp. 177-186.
(Ciobanu and Juravle, 2012) Ciobanu, G., and Juravle, C., 2012. Flexible Software Architecture and Language for Mobile Agents. Concurrency and Computation: Practice and Experience, 24(6), pp. 559-571.
(Clark et al., 2004) Clark, T., Evans, A., Sammut, P., and Willans, J., 2004. Language Driven Development and MDA. MDA Journal, pp. 2-13.
(Cossentino and Potts, 2002) Cossentino, M., and Potts, C., 2002. A CASE tool supported methodology for the design of multi-agent systems. In Proceedings of the International Conference on Software Engineering Research and Practice (SERP'02), Las Vegas.
(Depke et al., 2001) Depke, R., Heckel, R., and Kuster J. M., 2001. Agent-Oriented Modeling with Graph Transformations. Lecture Notes in Computer Science, 1957, pp. 105-119.
(vanDeursen et al., 2000) van Deursen, A., Klint, P., and Visser, J., 2000. Domain-specific languages: an annotated bibliography. ACM SIGPLAN Notices, 35 (6), pp. 26-36.
(Duddy et al., 2003) Duddy, K., Gerber A., Lawley, M., Raymond, K., and Steel, J., 2003. Model Transformation: A declarative, reusable patterns approach. In Proceedings of the 7th IEEE International Enterprise Distributed Object Computing Conference (EDOC'03), Brisbane, Queensland, Australia, pp. 174-185.
(Ferber, 1999) Ferber, J., 1999. Multi-agent systems: An introduction to distributed artificial intelligence. Addison-Wesley Professional, 528p.
(Ferber and Gutknecht, 1998) Ferber, J., and Gutknecht, O., 1998. A Meta-Model for the Analysis and Design of Organizations in Multi-Agent Systems. In Proceedings of the 3rd International Conference on Multi-Agent Systems, Paris, France, pp. 128-135.
(Finin et al., 1997) Finin, T., Labrou, Y., and Mayfield, J., 1997. KQML as an agent communication language, In Bradshaw (Ed): Software Agents, AAAI Press/MIT Press, pp. 291-316.
(FIPA, 2002a) Foundation for Intelligent Physical Agents (FIPA), 2002. FIPA Agent Communication Language Message Structure Specification, available at: http://www.fipa.org/specs/fipa00061/ (last access: March 2013).
(FIPA, 2002b) Foundation for Intelligent Physical Agents (FIPA), 2002. FIPA Agent Communication Language Specifications, available at: http://www.fipa.org/repository/aclspecs.html (last access: March 2013).
(FIPA Modeling TC, 2003) IEEE Foundation for Intelligent Physical Agents (FIPA) Modeling Technical Committee (Modeling TC), 2003. available at: http://www.fipa.org/activities/modeling.html (last access: March 2013).
(Fowler, 2011) Fowler, M., 2011. Domain-specific Languages. Addison-Wesley Professional, 640p.
(Freemind, 2002) Freemind, 2002. available at: http://freemind.sourceforge.net/wiki/index.php/Main_Page (last access: March 2013)
(Fuentes-Fernandez et al., 2010) Fuentes-Fernandez, R., Garcia-Magarino, I., Gomez-Rodriguez, A. M., Gonzalez-Moreno, J. C., 2010. A technique for defining agent-oriented engineering processes with tool support. Engineering Applications of Artificial Intelligence, 23(3), pp. 432–444.
(Gascuena et al., 2012) Gascuena, J. M., Navarro, E., and Fernandez-Caballero, A., 2012. Model-Driven Engineering Techniques for the Development of Multi-agent Systems. Engineering Applications of Artificial Intelligence, 25 (1), pp.159–173.
(Getir et al., 2011) Getir, S., Demirkol, S., Challenger, M., and Kardas, G., 2011. The GMF-based Syntax Tool of a DSML for the Semantic Web enabled Multi-Agent Systems. In Proceedings of the Workshop on Programming Systems, Languages, and Applications based on Actors, Agents, and Decentralized Control (AGERE! 2011), held at the 2nd Systems, Programming, Languages and Applications: Software for Humanity Conference (SPLASH 2011), Portland, USA, pp. 235-238.
(Google Co., 2002) Google Talk computer to computer voice and video chat software. available at: http://www.google.com/talk/ (last access: March 2013).
(GME, 2001) Ledeczi A., Maroti M., Bakay A., Karsai G., Garrett J., Thomason C., Nordstrom G., Sprinkle J.,Volgyesi P., 2001. The Generic Modeling Environment. In Proceedings of the Workshop on Intelligent Signal Processing. available at: http://www.isis.vanderbilt.edu/Projects/gme/ (last access: March 2013)
(Gray et al., 2007) Gray, J., Tolvanen, J-P., Kelly, S., Gokhale, A., Neema, S., and Sprinkle, J., 2007. Domain-Specific Modeling. In Fishwick (Ed): Handbook of Dynamic System Modeling, CRC Press, pp. 1-7.
(Haag et al., 2004) Haag, S., Cummings, M., McCubbrey, D. J., 2004. Management Information Systems for the Information Age. 4th ed., McGraw Hill.
(Hahn, 2008) Hahn, C., 2008. A Domain Specific Language for Multiagent Systems. In Proceedings of the 7th Autonomous Agents and Multiagent Systems Conference (AAMAS'08), Estoril, Portugal, pp. 233-240.

(Hahn and Fischer, 2009) Hahn, C., and Fischer, K., 2009. The Formal Semantics of the Domain Specific Modeling Language for Multi-agent Systems. Lecture Notes in Computer Science, 5386, pp. 145-158.

(Hahn et al., 2008) Hahn, C., Nesbigall, S., Warwas, S., Zinnikus, I., Fischer, K. and Klusch, M., 2008. Integration of Multiagent Systems and Semantic Web Services on a Platform Independent Level. In Proceedings of 2008 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT 2008), Sydney, Australia, pp. 200-206.

(Hahn et al., 2009) Hahn, C., Madrigal-Mora, C., and Fischer, K., 2009. A platform-independent metamodel for multiagent systems. Autonomous Agents and Multi-agent Systems, 18(2), pp. 239-266.

(Howden et al., 2001) Howden, N., Ronnquista, R., Hodgson, A., and Lucas, A., 2001. Jack intelligent agents: Summary of an agent infrastructure. In Proceedings of the 2nd International Workshop on Infrastructure for Agents, MAS, and Scalable MAS at the 5th International Conference on Autonomous Agents, Montreal, Canada.

(Huget, 2005) Huget, M-P., 2005. Modeling Languages for Multiagent Systems. In Proceedings of the 6th International Workshop on Agent-Oriented Software Engineering (AOSE 2005), Utrecht, the Netherlands

(JADEX, 2003) JADEX BDI Agent System, 2003. available at: http://jadex-agents.informatik.uni-hamburg.de/xwiki/bin/view/About/Overview (last access: March 2013).

(Jajah, 2006) Jajah Communication Solutions, 2006. available at http://www.jajah.com (last access: March 2013).

(Jena, 2011) Apache Jena, 2011. available at: http://jena.apache.org/ (last access: March 2013).

(Jouault and Kurtev, 2006) Jouault, F., and Kurtev, I., 2006. Transforming Models with ATL. Lecture Notes in Computer Science, 3844, pp. 128-138.

(Jouault et al., 2008) Jouault, F., Allilaire, F., Bezivin, J., and Kurtev, I., 2008. ATL: A model transformation tool. Science of Computer Programming, 72(1-2), pp. 31-39.

(Kalnins et al., 2005) Kalnins, A., Barzdins, J., and Celms E., 2005. Model Transformation Language MOLA. Lecture Notes in Computer Science, 3599, pp. 62-76.

(Kardas et al., 2009a) Kardas, G., Goknil, A., Dikenelli, O., and Topaloglu, N.Y., 2009. Model Driven Development of Semantic Web Enabled Multi-agent Systems. International Journal of Cooperative Information Systems, 18(2), pp. 261-308.

(Kardas et al., 2009b) Kardas, G., Ekinci, E. E., Afsar, B., Dikenelli, O., and Topaloglu, N. Y., 2009. Modeling Tools for Platform Specific Design of Multi-agent Systems. Lecture Notes in Artificial Intelligence, 5774, pp. 202-207.

(Kardas et al., 2010) Kardas, G., Demirezen, Z., and Challenger, M., 2010. Towards a DSML for Semantic Web enabled Multi-agent Systems. In Proceedings of the International Workshop on Formalization of Modeling Languages, held in conjunction with the 24th European Conference on Object-Oriented Programming (ECOOP 2010), Maribor, Slovenia, pp. 1-5.

(Kelly and Tolvanen, 2008) Kelly, S., and Tolvanen, J-P., 2008. Domain-Specific Modeling: Enabling Full Code Generation. John Willey & Sons, Inc., New Jersey, USA.

(Kos et al., 2011) Kos, T., Kosar, T., Knez, J. and Mernik, M., 2011. From DCOM interfaces to domain-specific modeling language: A case study on the Sequencer. Computer Science and Information Systems, 8(2), pp. 361-378.

(Kosar et al., 2012) Kosar, T., Carver, J., Mernik, M., 2012. Program Comprehension of domain-specific and general-purpose languages: comparison using a family of experiments. Empirical Software Engineering, 17(3), pp. 276-304.

(Kulesza et al., 2005) Kulesza, U., Garcia, A., Lucena, C., and Alencar, P., 2005. A Generative Approach for Multi-agent System Development. Lecture Notes in Computer Science, 3390, pp. 52-69.

(Li and Horrocks, 2003) Li, L., and Horrocks, I., 2003. A Software Framework for Matchmaking based on Semantic Web Technology. In Proceedings of the 20th International World Wide Web Conference (WWW 2003), Budapest, Hungary, pp. 331-339.

(Martin et al., 2004) Martin, D., Burstein, M., Hobbs, J. Lassila, O., McDermott, D., McIlraith, S. Narayanan, S. Paolucci, M., Parsia, B., Payne, T., Sirin, E., Srinivasan, N., and Sycara, K., 2004. OWL-S: Semantic Markup for Web Services. W3C Member Submission, available at: http://www.w3.org/Submission/OWL-S/ (last access: March 2013).

(Mernik, 2013) Mernik, M. (Ed), 2013. Formal and Practical Aspects of Domain-Specific Languages: Recent Developments. IGI Global.

(Mernik et al., 2005) Mernik, M., Heering, J., and Sloane, A., 2005. When and how to develop domain-specific languages. ACM Computing Surveys, 37(4), pp. 316-344.

(MetaCase, 1995) MetaCase, MetaEdit+ Domain-Specific Modeling (DSM) environment, 1995. available at: http://www.metacase.com/products.html (last access: March 2013).

(Microsoft, 2005) Microsoft DSL Tools, 2005. available at: http://www.microsoft.com/en-us/download/details.aspx?id=2379 (last access: March 2013)

(Molesini et al., 2005) Molesini, A., Denti, E. and Omicini, A., 2005. MAS Meta-models on Test: UML vs. OPM in the SODA Case Study. Lecture Notes in Artificial Intelligence, 3690, pp. 163-172.

(Nonoh, 2007) Nonoh VoIP calls, 2007. available at http://www.nonoh.net/ (last access: March 2013)

(Odell et al., 2005) Odell, J., Nodine, M., and Levy, R., 2005. A Metamodel for Agents, Roles and Groups. Lecture Notes in Computer Science, 3382, pp. 78-92.

(Oldevik et al., 2005) Oldevik, J., Neple, T., Gronmo, R., Aagedal J., and Berre, A. J., 2005. Toward Standardised Model to Text Transformations. Lecture Notes in Computer Science, 3748, pp. 239-253.

(OMG, 2000) Object Management Group, Unified Modeling Language Specification, 2000. available at: http://www.uml.org/ (last access: March 2013).

(OMG, 2002) Object Management Group, Meta Object Facility (MOF), 2002. available at: http://www.omg.org/spec/MOF/ (last access: March 2013)

(OMG, 2003) Object Management Group, Model Driven Architecture (MDA) Specification, 2003. available at: http://www.omg.org/mda/ (last access: March 2013)

(OMG, 2008) Object Management Group, Software Process Engineering Metamodel Specification Version 2.0, formal/2008-04-01, 2008. available at: http://www.omg.org/spec/SPEM/2.0/ (last access: June 2012)

(OMG, 2009) Object Management Group, Ontology Definition Metamodel (ODM), 2009. available at: http://www.omg.org/spec/ODM/1.0/ (last access: March 2013)

(OMG, 2012) Object Management Group, Object Constraint Language (OCL), 2012. available at: http://www.omg.org/spec/OCL/2.3.1/ (last access: March 2013)

(Omicini, 2000) Omicini, A., 2000. SODA: Societies and Infrastructures in the Analysis and Design of Agent-based Systems. Lecture Notes in Computer Science, 1957, pp. 185-193.

(Omicini et al., 2008) Omicini, A., Ricci, A., and Viroli, M., 2008. Artifacts in the A&A meta-model for multi-agent systems. Autonomous Agents and Multi-Agent Systems, 17(3), pp. 432-456.

(Oovoo, 2007) Oovoo Free video chats, 2007. available at: http://www.oovoo.com (last access: March 2013)

(Padgham and Winikoff, 2004) Padgham, L., and Winikoff, M., 2004. Developing Intelligent Agent Systems: A Practical Guide. John Wiley and Sons, 230p.

(Pavon et al., 2005) Pavon, J., Gomez-Sanz, J. J., Fuentes-Fernandez, R., 2005. The INGENIAS methodology and tools. In Henderson-Sellers, B., Giorgini, P. (Eds.), Agent-Oriented Methodologies, Article IX. Idea Group Publishing, pp. 236–276.

(Pavon et al., 2006) Pavon, J., Gomez-Sanz, J. J., and Fuentes, R., 2006. Model Driven Development of Multi-Agent Systems. Lecture Notes in Computer Science, 4066, pp. 284-298.

(Pokahr et al., 2005) Pokahr, A, Braubach, L, Lamersdorf, W., 2005. Jadex: A BDI Reasoning Engine. In Bordini et al. (Eds): Multi-Agent Programming Languages, Platforms and Applications, Springer, pp. 149-174.

(Pokahr et al., 2007) Pokahr, A., Braubach, L., Walczak, A., Lamersdorf, W., 2007. Jadex - Engineering Goal-Oriented Agents. In Bellifemine et al. (Eds): Developing Multi-Agent Systems with JADE, Wiley Publishing, pp. 254-258.

(Protégé, 2004) Protégé: Ontology Editor and Knowledge Acquisition System, 2004. available at: http://protege.stanford.edu/ (last access: March 2013).

(Rao and Georgeff, 1995) Rao, A., and Georgeff, M., 1995. BDI Agents: From Theory to Practice. In proceedings of the 1st International Conference on Multi-Agent Systems (ICMAS-95), San Francisco, pp. 312-319,

(Rougemaille et al., 2007) Rougemaille, S., Migeon, F., Maurel, C., and Gleizes, M-P., 2007. Model Driven Engineering for Designing Adaptive Multi-Agent Systems. Lecture Notes in Artificial Intelligence, 4995, pp. 318-332.

(Sánchez Cuadrado and Garcia Molina, 2007) Sánchez Cuadrado, J., and Garcia Molina, J., 2007. Building Domain-Specific Languages for Model-Driven Development. IEEE Software, 24(5), pp. 48-56.

(Schmidt, 2006) Schmidt, D.C., 2006. Guest Editor's Introduction: Model-Driven Engineering. IEEE Computer, 39(2), pp. 25-31.

(Shadbolt et al., 2006) Shadbolt, N., Hall, W., and Berners-Lee, T., 2006. The Semantic Web Revisited. IEEE Intelligent Systems, 21(3), pp. 96-101.

(Sprinkle et al., 2009) Sprinkle, J., Mernik, M., Tolvanen, J.-P., and Spinellis, D., 2009. Guest Editors' Introduction: What Kinds of Nails Need a Domain-Specific Hammer?. IEEE Software, 26(4), pp. 15-18.

(Smith, 1980) Smith, R.G., 1980. The Contract Net Protocol: High-level Communication and Control in a Distributed Problem Solver. IEEE Transactions on Computers, 29(12), pp. 1104-1113.

(Strembeck and Zdun, 2009) Strembeck, M., and Zdun, U., 2009. An approach for the systematic development of domain-specific languages. Software: Practice and Experience, 39(15), pp. 1253-1292.

(Sycara et al., 2003) Sycara, K., Paolucci, M., Ankolekar, A., and Srinivasan, N., 2003. Automated discovery, interaction and composition of Semantic Web Services. Journal of Web Semantics, 1(1), pp. 27-46.

(The Eclipse Foundation, 2005) The Eclipse Foundation, MOFScript model to text transformation language and tool, 2005. available at: http://www.eclipse.org/gmt/mofscript/ (last access: March 2013).

(The Eclipse Foundation, 2006) The Eclipse Foundation, Graphical Modeling Framework (GMF), 2006. available at: http://www.eclipse.org/modeling/gmp/ (last access: March 2013).

(The Eclipse Foundation, 2007a) The Eclipse Foundation, ATL Model Transformation Language and Toolkit, 2007. available at: http://www.eclipse.org/atl/ (last access: March 2013).

(The Eclipse Foundation, 2007b) The Eclipse Foundation, EMFT JET Editor, 2007. available at: http://www.eclipse.org/modeling/m2t/?project=jet#jet (last access: March 2013).

(Varanda-Pereira et al., 2008) Varanda-Pereira, M. J., Mernik, M., Da-Cruz, D., and Henriques, P. R., 2008. Program comprehension for domain-specific languages. Computer Science and Information Systems, 5(2), pp. 1-17.

(Vidal et al., 2001) Vidal, M., Buhler, P. A., and Huhns, M. N., 2001. Inside an Agent. IEEE Internet Computing, 5(1), pp. 82-86.

(W3C, 1999) World Wide Web Consortium, Resource Description Framework, 1999. available at: http://www.w3.org/RDF/ (last access: March 2013).

(W3C, 2004) World Wide Web Consortium, OWL Web Ontology Language, 2004. available at: http://www.w3.org/TR/owl-features/ (last access: March 2013).

(Warmer and Kleppe, 2003) Warmer, J. and Kleppe, A., 2003. Object Constraint Language: The Getting Your Models Ready for MDA, 2nd Edition, Pearson Education, USA, 240p.

(Warwas and Hahn, 2008) Warwas, S., and Hahn, C., 2008. The concrete syntax of the platform independent modeling language for multiagent systems. In Proceedings of the Agent-based Technologies and applications for enterprise interoperability, held in conjunction with the 7th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2008), Estoril, Portugal.

(Wooldridge and Jennings, 1995) Wooldrige, M., and Jennings, N. R., 1995. Intelligent agents: theory and practice. The Knowledge Engineering Review, 10(2), pp. 115-152.

(WSDL-S, 2005) WSDL-S: Web Service Semantics, 2005. available at: http://www.w3.org/Submission/WSDL-S/ (last access: March 2013).

(WSMO, 2005) WSMO: Web Service Modeling Ontology, 2005. available at: http://www.w3.org/Submission/WSMO/ (last access: March 2013).

(Zambonelli et al., 2003) Zambonelli, F., Jennings, N. R., and Wooldrige, M., 2003. Developing multiagent systems: The Gaia methodology. ACM Transactions on Software Engineering and Methodologies, 12(3), pp. 317-370.