



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



Moharram Challenger^{a,*}, Sebla Demirkol^a, Sinem Getir^a, Marjan Mernik^b,
Geylani Kardas^{a,*}, Tomaž Kosar^b

^a International Computer Institute, Ege University, Bornova, 35100 Izmir, Turkey

^b Faculty of Electrical Engineering and Computer Science, University of Maribor, Smetanova 17, 2000 Maribor, Slovenia

ARTICLE INFO

Article history:

Received 3 June 2012

Received in revised form

8 November 2013

Accepted 19 November 2013

Available online 21 December 2013

Keywords:

Agent

Multiagent system

Model driven development

Domain-specific modeling language

Metamodel

Semantic web

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 called SEA_ML 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 methodology proposed for the MAS development based on SEA_ML is also discussed including its example application on the development of an agent-based stock exchange system.

© 2013 Elsevier Ltd. All rights reserved.

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.

* Corresponding author at: International Computer Institute, Ege University, Bornova, 35100 Izmir, Turkey. Tel: +90 232 311 3223, Fax: +90 232 388 7230.

E-mail addresses: moharram.challenger@mail.ege.edu.tr, m.challenger@gmail.com (M. Challenger), sebla.demirkol@ege.edu.tr (S. Demirkol), sinem.getir@ege.edu.tr (S. Getir), marjan.mernik@uni-mb.si (M. Mernik), geylani.kardas@ege.edu.tr (G. Kardas), tomaz.kosar@uni-mb.si (T. Kosar).

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 García 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 (the 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 Section 4 and Section 5, respectively. Section 6 discusses the methodology proposed for the MAS development based on SEA_ML, including its example application on the development of an agent-based stock exchange system. 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 (Figs. 1–8) 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.

The collection of SEA_ML viewpoints constitutes an extensive and all-embracing model of the MAS domain. The SEA_ML views, namely MAS, Agent Internal and Interaction, cover the main agent entities and their relations on which the agent research community is mostly agreed. Similar abstractions can also be found in e.g. Pavon et al. (2006), Hahn et al. (2009), Beydoun et al. (2009) as will be broadly discussed in the related work section of this paper. Furthermore, more specific aspects of the domain like Plan, Role and Environment are also supported in SEA_ML syntax with individual metamodels. In fact, proposing specific metamodels for these views is not a new idea. For example, Hahn (Hahn, 2008) also presented sub-metamodels for the Plan and Role aspects while the Environment view was specifically considered in Omicini et al. (2008) and Challenger et al. (2011). SEA_ML's abstract syntax combines all of these generally accepted aspects of MAS and introduces two new viewpoints (Agent-SWS Interaction and Ontology) for supporting the development of software agents working within the Semantic Web environment. On the other hand, the behavior and goal aspects can also be considered with distinct sub-metamodels although such an approach is not very common in MAS DSML proposals. Due to the generality considerations and the scarcity of meta-entities for being a unique model, SEA_ML does not consider these aspects in individual sub-metamodels. Instead, the required first class entities and their relations pertaining to the behavior and goal aspects of agent systems are included within the agent internal viewpoint of SEA_ML like many other MAS DSMLs.

It is also worth indicating that grouping MAS domain entities according to different views also facilitate the evolution of the SEA_ML's syntax in addition to easy interpretation and usage by the developers. For instance, updating specific SEA_ML viewpoints (e.g. including,

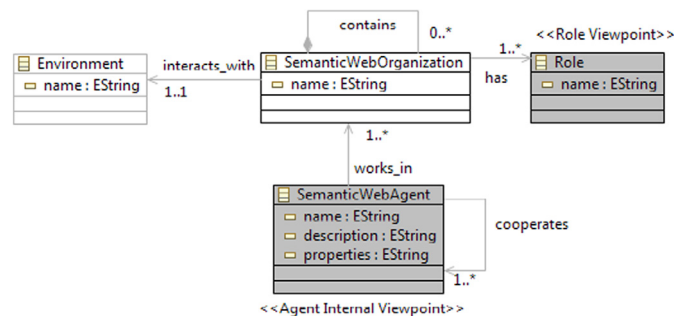


Fig. 1. MAS viewpoint.

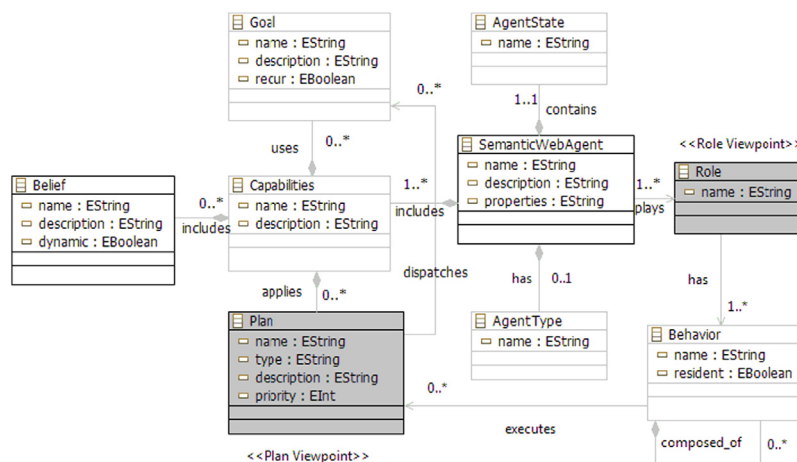


Fig. 2. Agent Internal viewpoint.

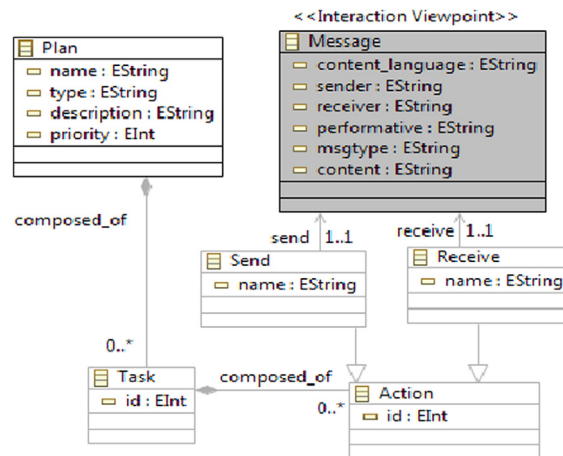


Fig. 3. Plan viewpoint.

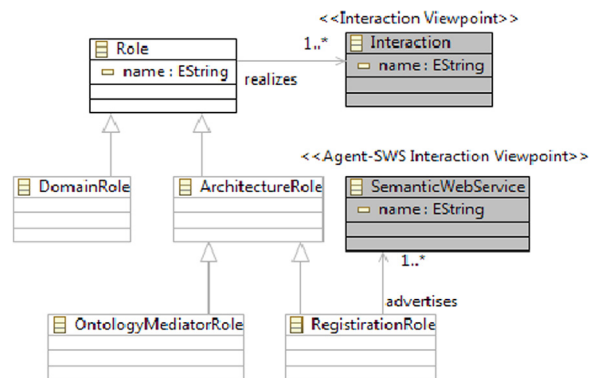


Fig. 4. Role viewpoint.

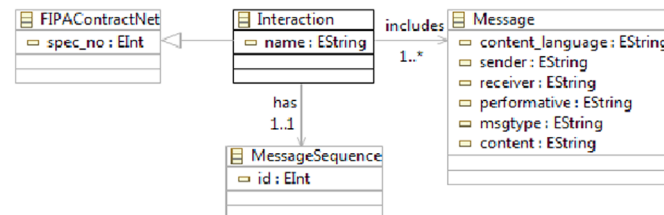


Fig. 5. Interaction viewpoint.

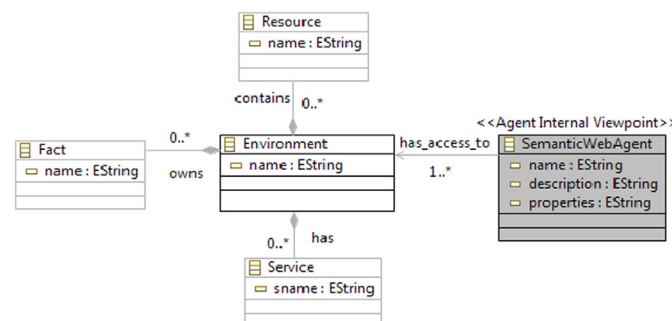


Fig. 6. Environment viewpoint.

removing or modifying entities and relations) is definitely much more comfortable than dealing with a huge metamodel with a single viewpoint. Further, with this multiple view structure extension of the current syntax is also possible for the developers, in case of some specific needs in the future.

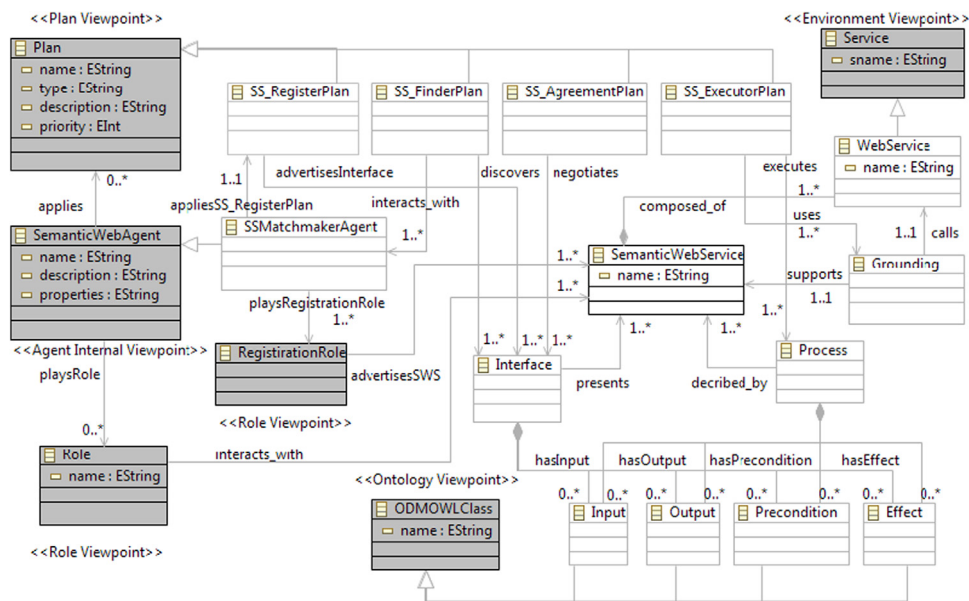


Fig. 7. Agent-SWS Interaction viewpoint.

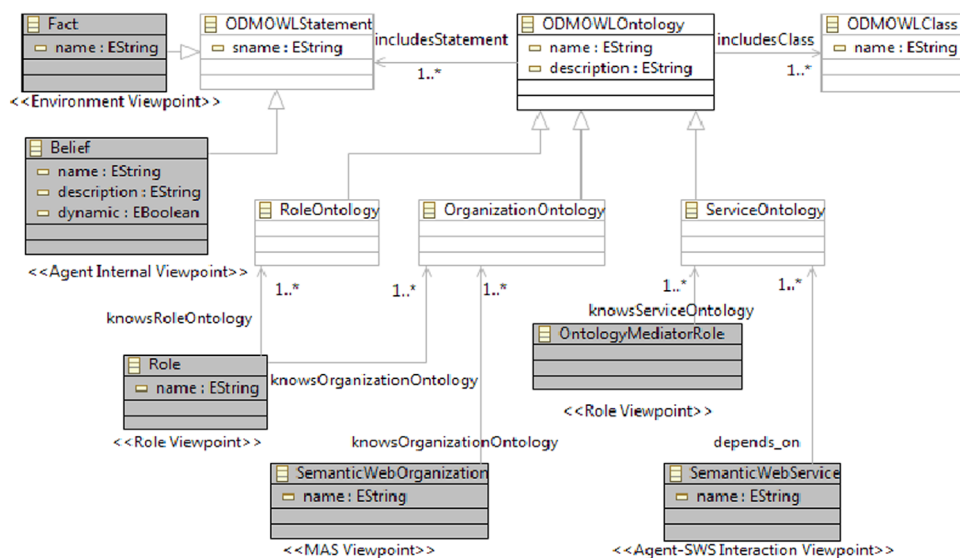


Fig. 8. Ontology viewpoint.

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 Fig. 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 Fig. 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 (Fig. 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 Section 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.

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 Fig. 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.

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 Fig. 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.

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 Section 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 Fig. 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.

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 Fig. 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 acknowledgment. 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.

2.6. Environment viewpoint

The Environment viewpoint, Fig. 6, focuses on the relations between agents and to what they access. The *Environment*, in which agents reside, contains all non-agent *Resources* (e.g. database, network device), *Facts* and *Services*. 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.

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, *SemanticWebAgents* apply *Plans* 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 (Fig. 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 Fig. 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. 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 Fig. 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.

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* (Fig. 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) (Ledeczi et al., 2001) which is based on a set of built-in generic concepts. It is also extensible and can be used for the General Purpose Languages (GPLs) 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](#)).

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 DSLs. 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	
Semantic service finder plan	
Semantic service agreement plan	
Semantic service executor plan	
Role	
Registration role	
Semantic web service	
Process	
Interface	
Grounding	

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	

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.

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".

3.1. 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 follows:

3.1.1. 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.

3.1.2. 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.

3.1.3. 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.

3.1.4. 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.

3.2. 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:

3.2.1. 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. Fig. 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.

3.2.2. 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.

3.2.3. 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.

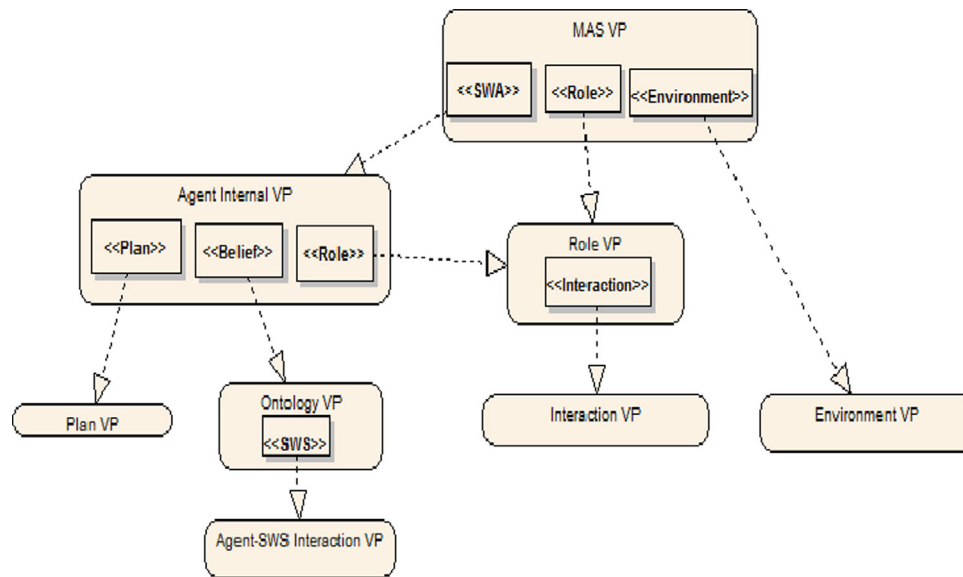


Fig. 9. System integration and transitions between SEA_ML viewpoints.

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.

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 (PSMMs) 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 (ADFs). 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.

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.

Listing 1. Definition of metamodels for ATL rules.

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

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”.

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

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

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.

Listing 2. Transformation from SEA_ML SemanticWebAgent to JADEX Agent.

```

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      }
```

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.

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.

Listing 3. *part1PatternforSWA* helper rule.

```

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

```

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.

Listing 4. *executorPlan* helper rule.

```

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

```

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_ExecutorPlans 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.

Listing 5. *part1PatternforService* helper rule.

```

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

```

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.

Listing 6. *part1PatternforInterface* helper rule.

```

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

```

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

Listing 7. *part1PatternforBehavior* helper rule.

```

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

```

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.

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 4–8. 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.

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

```

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

```

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 >

```

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.

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

```

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      }

```

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. MAS development methodology based on SEA_ML

SEA_ML's features discussed in [Sections 2–5](#) are combined to constitute a model-driven MAS development methodology in which MAS developers can visually design and implement their agent systems for various agent deployment platforms. The proposed MAS methodology based on SEA_ML includes three main steps (see [Fig. 10](#)) following each other: 1. System modeling, 2. Automatic transformation of the models, and finally 3. Code generation for exact MAS implementation.

In the system modeling step the agent developer uses the fully functional graphical editors of SEA_ML to sketch the system, which includes 8 viewpoints of SEA_ML's syntax. Each viewpoint has its own palette as described in [Section 3](#). Although, this step may seem to imply a system design in traditional software development methodologies, the tool not only offers a computer-aided design (CAD) for

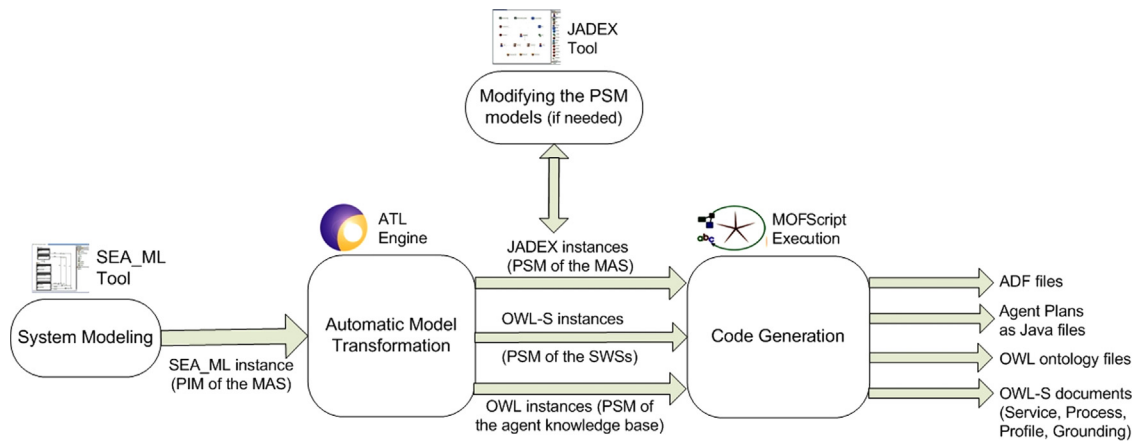


Fig. 10. SEA_ML-based MAS development methodology.

system modeling, but also provides various controls (originating from SEA_ML's concrete syntax constraints) which lead the designers into providing more accurate models. The result of this step is the developed platform independent (SEA_ML) model of the MAS.

The next step in the MAS development methodology based on SEA_ML is the automatic model transformation. The models created in the previous step should be transformed from platform independent level into the platform-specific level, in this case the JADEX and OWL-S models. The source models for these transformations conform to SEA_ML's metamodel as discussed in Section 2 and they are generic enough to be transformable automatically into different platform-specific models. These transformations are called M2M transformations, as mentioned previously. The target models, achieved after the automatic application of the SEA_ML M2M transformation rules, conform to the JADEX and OWL-S metamodels. The transformation rules written in ATL are discussed in Section 4. These transformation rules are integrated within the SEA_ML's tool and in fact the agent developer does not care about the context of the rules. Following the creation of models in the previous modeling step, the only thing requested from the developers is to choose the specific menu options for the automatic execution of these transformation rules on their platform independent MAS models. The obtained target models will preserve the syntax of the target languages. At this stage, the developers have two options: (1) they may directly continue with the last step; code generation for the achieved platform independent MAS models or (2) if they wish, they may visually modify the achieved target models in order to elaborate or customize them, which in fact will lead to the gaining of a more accurate software code in the next step. For this purpose, the graphical modeling environment for the JADEX BDI agents (Kardas et al., 2009b) can be used collaboratively with the SEA_ML's MAS development tool, as discussed at the end of Section 4.

The last step of the SEA_ML-based development methodology is the software code generation for MAS. At the beginning of this step, the agent developers own platform-specific agent software models conform to the target MAS deployment frameworks/languages. Now it is time to generate software codes for the models, in these target languages. In this way, the developers' high level initial models (in SEA_ML) are converted into the code in the target language. The M2T transformation rules written in MOFScript (discussed in Section 5) are automatically executed on the target MAS models and codes are obtained for the implementation of the MAS. Similar to the M2M transformation step, the agent developer does not need to know about both the context of the rules and their execution details. The developer only selects the code generation feature over the SEA_ML tool's GUI. The result of the automatic code generation step will be the JADEX files (including XML encoded ADF files and agent plans as Java files) and OWL-S documents (including Service, Profile, Process and Grounding documents). Based on the initial models of the developer, these files are also interlinked during the transformations where it is required.

Finally, it is worth noting that, all of the three steps in this MAS development methodology are supported by the SEA_ML's graphical tool. The agent developer models the desired MAS via the provided GUI and all the remaining development activities pertaining to the above discussed steps can be performed automatically within the development environment of the tool. However, at any stage, the developer may intervene in this development process if he/she wishes to elaborate or customize the achieved artifacts.

In the following subsection, the development of an agent-based stock exchange system is discussed in order to both evaluate and provide some flavor of the use of this MDD methodology based on SEA_ML.

6.1. Case study: development of an agent-based stock exchange system

Stock trading is one of the key items in an economy. There are many participators, human or non-human, working together to realize the trading in this system. Some of these are investors, brokers, companies, stocks, trade floors, intermediate traders, stock markets, and so on. Automating the interactions of these systems could make the trading more efficient. However, the software required for this automation is fairly complex. In order to deal with this complexity and to have the ability of adding intelligent entities to the work, software agents can be used instead of human beings. Various studies exist (e.g. El-Sawi and Ali, 1998; Luo et al., 2002; Kendall and Su, 2003; Posada, 2006; Rahimi et al., 2009; Kardas et al., 2012) which propose agent-based solutions to cope with the challenges of a stock exchange. By considering the properties of autonomous agents (autonomous, responsive, proactive, and sociability), the use of MASs both in modeling stock trading markets and exact implementation of software systems in real world stock trading applications is worthwhile.

Perhaps one of the important issues, which should be taken into account, during the realizations of agent-based stock trading systems, is the appropriate modeling of agents' behavior and their interaction with each other and with (semantic) web services within a software system. The execution order of an agent's plans is to find, negotiate and realize a trade that is mostly based on known facts about the environment and these facts dynamically change during the lifecycle of a software agent.

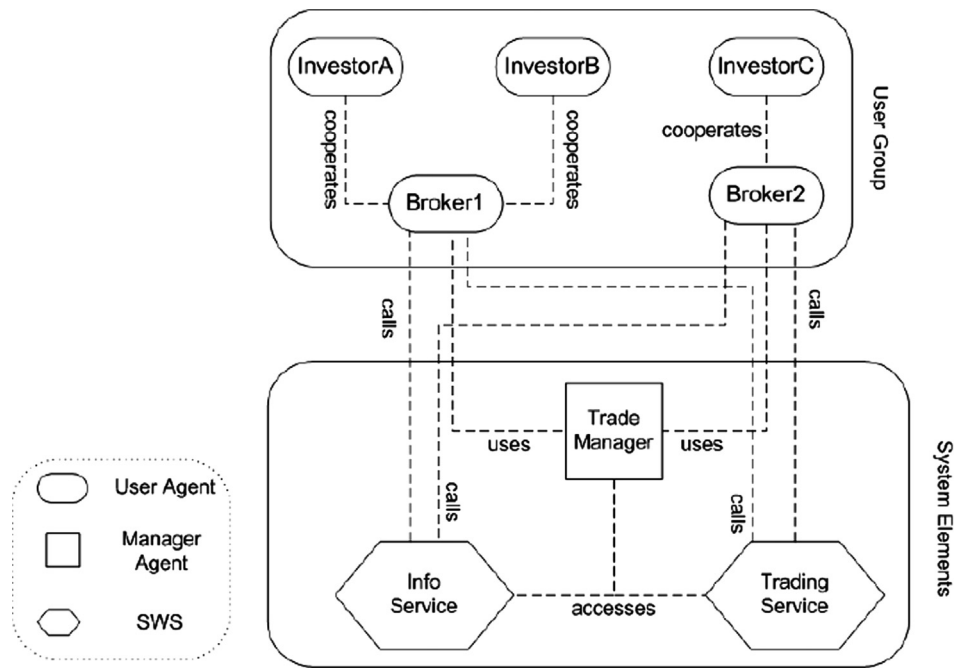


Fig. 11. Architecture of the Stock Exchange case study.

As a motivating example, consider the following scenario in which a user (Investor) orders the buying of some stocks. The user should be able to have access to the trade information service. He/she should also be able to consult with a broker and find the proper stock and a seller. Finally, an order should be issued by the user considering certain factors like rate of exchange or the oil fee on the market. However, these tasks could be complex and cumbersome for human beings to deal with. Different types of agents might represent the investors, brokers, trade manager, and so on. Further, the necessary services could be represented using semantic web services constructed in such a way that they could be accessed and interacted by agents. Hence, the tasks of communication and interaction between many elements of the stock exchange system could be performed automatically, accurately and quickly by software components instead of human users.

Let us assume that some web services exist for finding, negotiating, and exchanging stocks on the trading floor of a stock exchange market. The trading floor performs transactions in which bids and demands are matched according to specified rules. Services in this case they would work on the web and use ontologies to execute these tasks on behalf of their human users by taking into consideration QoS metrics. On the other hand, the agents represent brokers and investors who intend to issue an order for the selling or buying a stock. These agents collaborate with a trade manager agent to realize interaction with the services in order to carry out the orders of their owner. The architecture of such an agent-based stock trading system is depicted in Fig. 11.

This architectural design considers that the owners of InvestorA and B are working on the same kinds of stock, say oil and gas. In this scenario, Broker1 is the expert on oil and gas investments. One happy outcome for this scenario could be that InvestorA wants to sell his or her oil stock and in the meantime InvestorB wants to buy the same types of stock. On the other hand, Investor C is going to invest in the food industry for which Broker2 has expertise. According to the conventions in the Stock Exchange market, the investors consult with brokers and monitor the trade market info and then the order is issued by Brokers to the market (in our design by the mediation of a Trade Manager).

6.1.1. System modeling

Conforming to the SEA_ML-based MDD methodology, we start by creating system models according to different viewpoints. In this example, knowledge for designing these models is gathered as a result of proper requirement engineering within the domain of the Stock Exchange System. The modeling and development of our system are based on the exact trading procedures of the Istanbul Stock Exchange (ISE) (ISE, 2013), which is one of the more important exchanges in the world. Interested users may also refer to Kardas et al. (2012) for an extensive discussion on the agent-based realization of ISE's trading procedures.

For this case study, we focused on three main viewpoints, namely, MAS, Agent internal, and Agent-SWS interaction viewpoints of the stock exchange system. The diagrams representing the models for these viewpoints are depicted in Figs. 12–14.

According to the scenario mentioned above, we modeled the interactions and the trading process of the system using SEA_ML. We considered all SWA agent instances within the MAS viewpoint aspect and then evaluated each agent's internal structure within the Agent Internal viewpoint. After that, we modeled these agents' interactions using semantic web services and web service internal components from the Agent-SWS interaction viewpoint.

First, an overview of the system was modeled using the MAS viewpoint, see Fig. 12. When considering the structure of the system, the semantic web agents of this case study worked within a semantic web organization named StockSystem. This main organization included two sub-organizations, StockUsers where the Investor and Broker agents reside, and the StockMarket where the system's internal agents, e.g. TradeManager (a SSMatchmaker agent instance) work. The StockMarket organization also has two sub-organizations, the TradingFloor and the StockInfoSystem. Some of these organizations and sub-organizations have their own organizational roles. For example, the superior organization StockSystem has the general roles of Exchanging, while the agents in StockMarket sub-organization play the Managing roles, and agents in the StockUsers sub-organization play the Interacting roles. These organizations also need to access

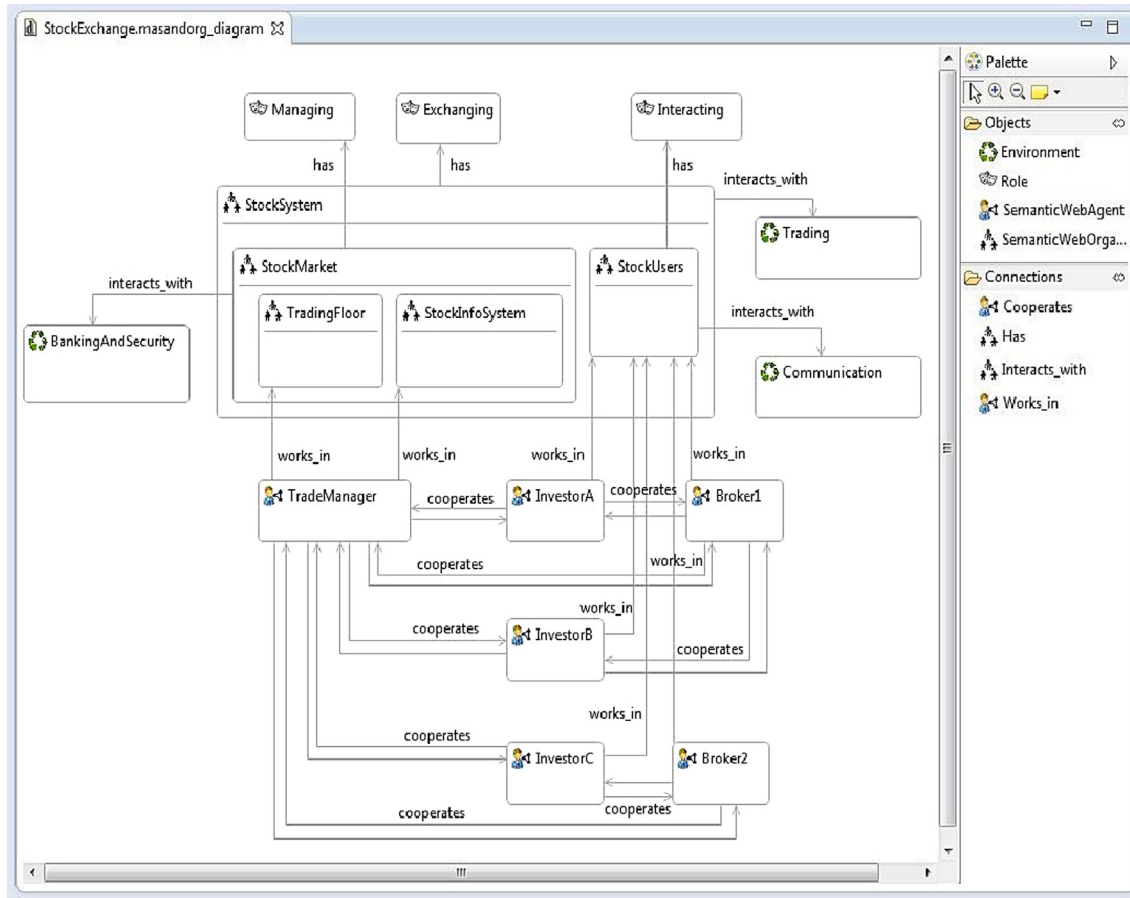


Fig. 12. Graphical modeling for the MAS viewpoint of the Stock Exchange system in the SEA_ML graphical syntax tool.

some resources in other environments. Therefore, they have interactions with the required environments in order to gain access permissions. For example, agents in the StockMarket sub-organization need to access bank accounts and some security features, so that they can interact with the BankingAndSecurity environment. In a similar way, agents of the StockSystem organization interact with the Trading environment and StockUsers interact with the Communication environment. In addition, the interactions of agents with each other were also modeled. MAS for this case study was composed of 6 semantic web agent instances, 1 trade manager, 2 brokers, and 3 investors. All of the user agents including Investors and Brokers cooperate with TradeManager to access the StockMarket. Also, the user agents interact with each other. For instance, InvestorA and InvestorB cooperate with Broker1 in order to exchange the stock for which Broker1 is an expert. Similarly, InvestorC refers to Broker2 to ask for his/her expertise (Fig. 12).

After overall modeling of the system, the internal structures of each semantic web agent were modeled. Fig. 13 illustrates an instance model of the agent internal viewpoint for an investor agent, called InvestorA. The instance model covered all of the required roles, behaviors, plans, goals, and beliefs of the agent. It also contains all the plan types which would be used in the Agent–SWS interaction viewpoint for discovering, negotiating with, and executing the candidate services.

InvestorA agent had a Capability called Stock including its Goals (“Sell available stock” and “Buy new stock”), Beliefs (“My business history”, and “Market Fluctuation Knowledge”), and Plans (StockFinder, and InvestorFinder). When considering the Beliefs, the agent would need them when collecting of certain facts (environmental facts and personal facts) in order to decide about the order of selling or buying. The agent could also play Selling and Buying roles. The Selling role could realize its task over “Finding a buyer” behavior by calling the InvestorFinding plan. Similarly, in the Buying role, the agent could adopt the “Finding a seller” behavior by calling the StockFinder plan. Finally, this diagram includes some information about the agent itself like agent type and state. In this case, the agent type would be “Business and Trading” and is initiated within an Active state.

Fig. 14 shows a screenshot of the instance model from the Agent–SWS interaction viewpoint, with including semantic services and the required plan instances. Investor and Broker agents could be modeled with appropriate plan instances in order to find, make the agreement with and execute the services which are the instances of the SS_FinderPlan, SS_AgreementPlan, and SS_ExecutorPlan, respectively. The services could also be modeled for interaction between the semantic web service’s internal components (such as Process, Grounding, and Interface), and the SWA’s plans.

So, when considering an InvestorA’s request for Buying some stocks, the agent would play the Buying role and apply its StockFinder plan for finding an appropriate Trading service interface of one TradingService SWS. This plan would realize the discovery by interacting with the TradeManager SSMatchmaker agent which has registered the services by applying the StockRecorder plan. As InvestorA cooperates with Broker1 in order to receive some expert advice for its investment, at the next step, the Broker1 agent applies its StockBargaining plan for negotiating with the already discovered services. This negotiation would be made through the Trade interface of the SWS. Finally, if the result of the negotiation was positive, the agent would apply the StockOrder plan to call the TradingFloor of the SWS by executing its Exchange process and using its TradeAccess grounding with which the service would be realized. In a similar way,

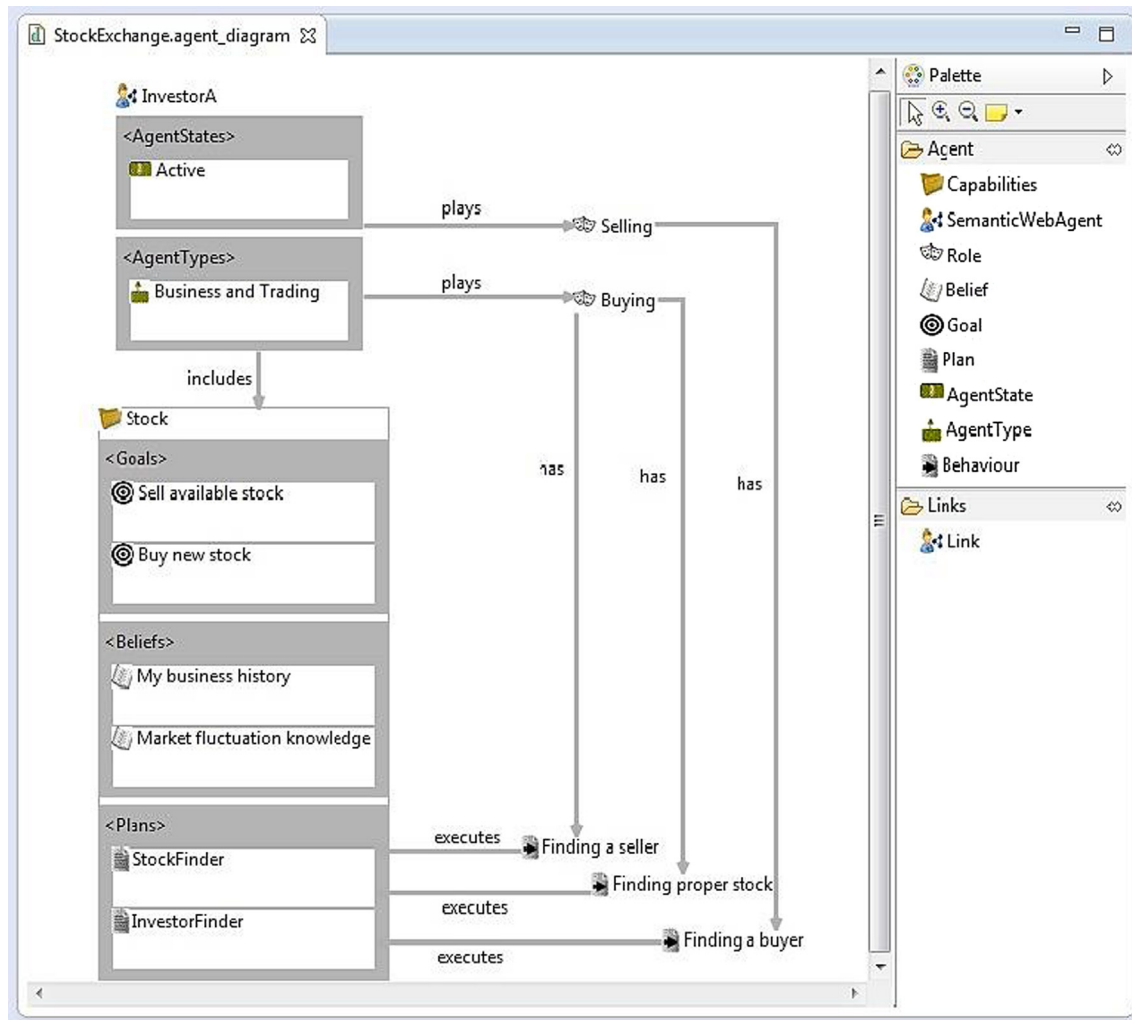


Fig. 13. Graphical modeling for the agent internal viewpoint of a multi-agent stock exchange system in the graphical syntax tool of SEA_ML.

Investor agents could cooperate with Brokers and interact with the TradeManager in order to collect some information about the market, e.g. the rate of exchange for a currency or the fluctuation rate for a specific stock.

6.1.2. Automatic transformation of the models

From the user point of view, the system modeling step needs more effort than other steps, since it should convey the result of the system analysis to the system design by providing the instance models in SEA_ML's GUI tools. However, the developer can put the necessary effort on extending or modifying the generated intermediate models. It is clear that the generated architectural code should be completed by the developer at the end.

The next step after modeling is transforming these models (PIM level models) into the models of target language (PSM level models). This is performed by simply executing the ATL rules discussed in Section 4. After executing the M2M transformation rules on the instance models, two types of models are provided, JADEX instance models and OWL-S instance models. These intermediate models represent our models in the target language. For example, the JADEX counterpart model for the agent internal viewpoint of our case study is illustrated in Fig. 15.

If desired, the developer can modify this model by adding/updating the components which would result in a more enriched code in the next step. For this purpose, 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 to generate the code.

Based on the mappings in Table 5 for model transformation, the elements in the Agent Internal metamodel are mapped to the elements in the JADEX metamodel. Therefore, an automatic application of the ATL rules creates elements; for example "Sell available stock" and "Buy new stock" goals for the Stock Exchange system, see Fig. 15. Similarly, the agent's plans like StockFinder and InvestorFinder plans are created, as shown in Fig. 15.

One type of transformation is used for each of the Agent Internal viewpoints and the MAS viewpoint separately (SEA_ML to JADEX), while two types of ATL rules are executed for the Agent–SWS Interaction viewpoint: one for the multi-agent part of the system (JADEX agents) which adds the 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 15 shows the OWL-S instance target model generated after automatic application of ATL transformation rules for the Agent–SWS Interaction viewpoint of the case study. The model elements are presented in XMI format. For example, the TradingService and InfoService semantic web services are defined in Lines 4 and 19 respectively. Process, Interface and Grounding are defined for TradingService SWS in Lines 5, 11, and 17; and for InfoService SWS in Lines 20, 26, and 32.

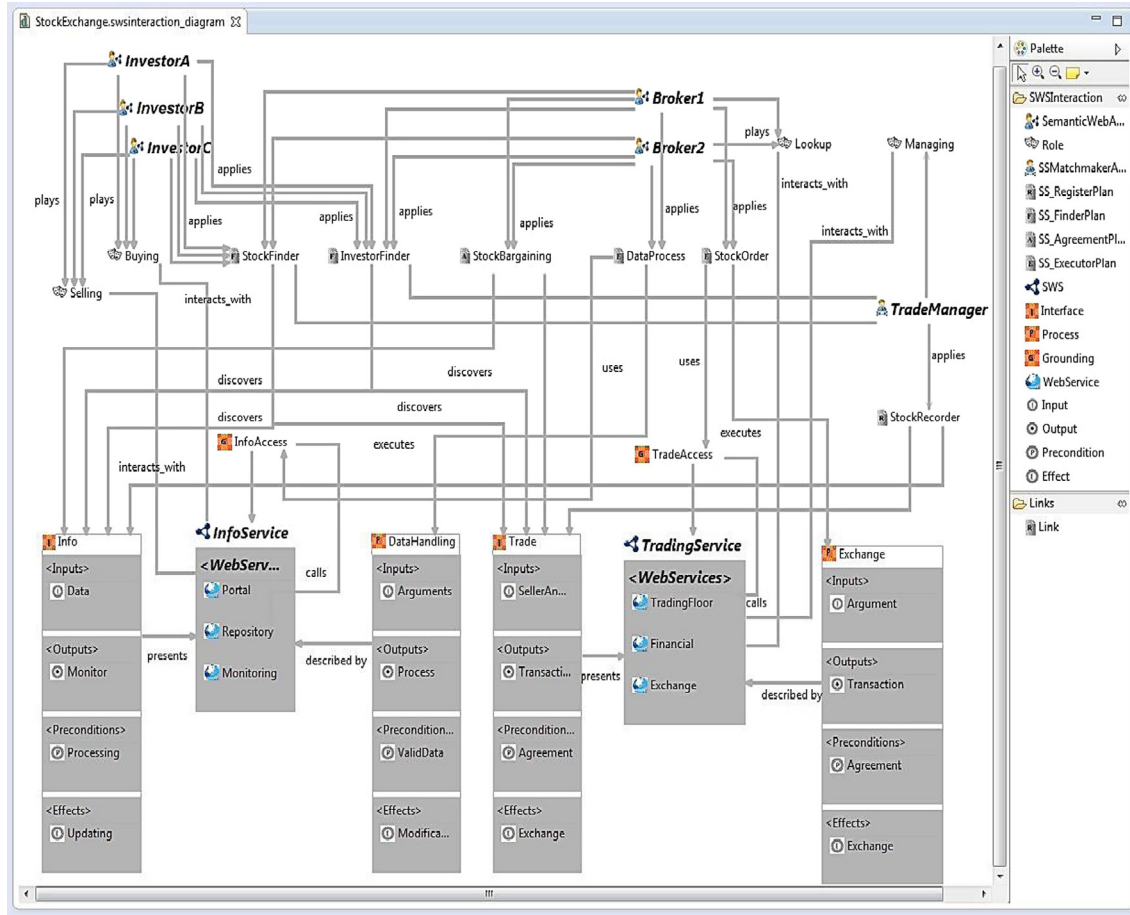


Fig. 14. Graphical modeling for the agent-SWS Interaction viewpoint of a multi-agent stock exchange system in the graphical syntax tool of SEA_ML.

Listing 15. Part of the generated OWL-S instance model for the Agent–SWS-Interaction viewpoint of the Stock Exchange system.

```

01 < ?xml version="1.0" encoding="ISO-8859-1"? >
02 < owls:OWLSplatform xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
03   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:owls="http://owls.com" >
04   < containsService name="TradingService" >
05     < describes name="Exchange"/ >
06       < containsInput name="argument"/ >
07       < containsOutput name="transaction"/ >
08       < containsCondition name="Agreement"/ >
09       < containsEffect xsi:type="owls:ResultVar" name="exchange"/ >
10     < /describes >
11     < presentedBy name="Trade"/ >
12       < containsInput name="sellerAndBuyer"/ >
13       < containsOutput name="transaction"/ >
14       < containsCondition name="Agreement"/ >
15       < containsEffect xsi:type="owls:ResultVar" name="exchange"/ >
16     < /presentedBy >
17     < supportedBy name="TradeAccess"/ >
18   < /containsService >
19   < containsService name="InfoService" >
20     < describes name="DataHandling"/ >
21       < containsInput name="arguments"/ >
22       < containsOutput name="process"/ >
23       < containsCondition name="ValidData"/ >
24       < containsEffect xsi:type="owls:ResultVar" name="updating"/ >
25     < /describes >
26     < presentedBy name="Info"/ >
27       < containsInput name="data"/ >
28       < containsOutput name="monitor"/ >
29       < containsCondition name="Processing"/ >

```

```

30         < containsEffect xsi:type="owls:ResultVar" name="modifications" />
31     < /presentedBy >
32     < supportedBy name="InfoAccess" />
33 < /containsService >
34 ...
35 < /owls:OWLSplatform >

```

6.1.3. Code generation for MAS implementation

Finally, the related software codes would be generated after modeling the MAS by using the SEA_ML's graphical tool and achieving the intermediate models. The previously discussed model-to-code transformations (Section 5) would be used in order to do this.

Artifacts gained as a result of the work on the Agent Internal viewpoint and the Agent–SWS Interaction viewpoint are discussed below in order to exemplify the generated code by the application of written MOFScript rules. The first one generates JADEX files including ADFs (XML) and Plan (Java) files. The second one not only generates JADEX files, but also generates OWL-S documents.

When the developer requests code generation for the platform-specific models, M2T transformations, written in MOFScript are automatically executed on these models and an ADF file for each agent, e.g. InvestorA, and a plan file (Java class) for each Plan element of the agent are generated. When considering our case study, part of the generated ADF file for InvestorA agent is given in Listing 16.

In Listing 16, all of the meta-elements and their attributes correspond with the related tags of JADEX ADF. The InvestorA agent's capability, beliefs, and goals are defined in Lines 6, 8–17, and 18–35. All the attributes of the JADEX metamodel are not included in the SEA_ML metamodel. Therefore, lines 28–30 are the default values of the corresponding attributes. In order to prevent repetition, some parts of the generated code are not shown. StockFinder and InvestorFinder plans (Lines 37 and 38) are the SS_FinderPlans of the Investor agent for looking up proper TradingService and InfoService SWSs.

Listing 16. An excerpt from the generated ADF file for the Agent Internal viewpoint of InvestorA agent in Stock Exchange case study.

```

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 http://jadex.sourceforge.net/jadex-2.0.xsd"
03 name="InvestorA" package="" description="" propertyfile="jadex.config.runtime" abstract="false" >
04     < imports > < /imports >
05     < capabilities >
06     < capability name="Stock" file="" description="" < /capability >
07 < /capabilities >
08     < beliefs >
09     < beliefset name="My business history" class="" exported="false"
10 description="" ubdaterate="0" transient="false" dynamic="false" >
11     < fact > "" < /fact >
12 < /beliefset >
13     < beliefset name="Market fluctuation knowledge" class="" exported="false"
14 description="" ubdaterate="0" transient="false" dynamic="false" >
15     < fact > "" < /fact >
16 < /beliefset >
17 < /beliefs >
18     < goals >
19     < achievegoal name="Sell available stock" recur="false" description=""
20 exclude="when_tried" exported="false" posttoall="false"
21 randomselection="false" recalculate="true"
22 recurdelay="0" retry="true" retrydelay="0" >
23     < creationcondition > < !- Write Creation Condition - > < /creationcondition >
24     ...
25     < failurecondition > < !- Write Failure Condition - > < /failurecondition >
26 < /achievegoal >
27     < achievegoal name="Buy new stock" recur="false" description=""
28 exclude="when_tried" Exported="false" posttoall="false"
29 randomselection="false" recalculate="true"
30 recurdelay="0" retry="true" retrydelay="0" >
31     < contextcondition > < !- Write Context Condition - > < /contextcondition >
32     ...
33     < dropcondition > < !- Write Drop Condition - > < /dropcondition >
34 < /achievegoal >
35 < /goals >
36 < plans >
37     < plan name="StockFinder" description="" exported="false" priority="0" > < /plan >
38     < plan name="InvestorFinder" description="" exported="false" priority="0" > < /plan >
39 < /plans >
40 ...
41 < /agent >

```

When applying the transformation rules for Agent–SWS Interaction viewpoint, nine ADF files (for the agents and their roles) and five plan files (for all agents) are generated when considering the Investor agents, Broker agents, TradeManager agent, and their roles. 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, 795 lines of code (LOC) were 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 InverstorA agent.

An excerpt from an OWL-S Service file is given as an example of the generated SWS files in Listing 17. The “Service.owl” file includes references to the Service Profile, the Service Model, and the Service Grounding files, which are generated for the TradingService SWS. Lines 39, 41 and 43 of Listing 17 cover these references for the Profile, the Process Model, and the Grounding of the SWS respectively.

Listing 17. An excerpt from the generated OWL-S Service file (“Service.owl”) for TradingService SWS

```

01 <?xml version="1.0" encoding="ISO-8859-1"? >
02 <!DOCTYPE urdef [
03   <!ENTITY rdf "http://www.w3.org/1999/02/22-rdf-syntax-ns" >
04   <!ENTITY rdfs "http://www.w3.org/2000/01/rdf-schema" >
05   <!ENTITY owl "http://www.w3.org/2002/07/owl" >
06   <!ENTITY service "http://www.daml.org/services/owl-s/1.0/tradingService.owl" >
07   <!ENTITY congo_profile "http://www.daml.org/services/owl-s/1.0/tradingServiceProfile.owl" >
08   <!ENTITY congo_process "http://www.daml.org/services/owl-s/1.0/tradingServiceProcess.owl" >
09   <!ENTITY congo_grounding "http://www.daml.org/services/owl-s/1.0/tradingServiceGrounding.owl" >
10   <!ENTITY DEFAULT "http://www.daml.org/services/owl-s/1.0/tradingService.owl" > ]>
11 <rdf:RDF
12   xmlns:rdf="&rdf;#"
13   xmlns:rdfs="&rdfs;#"
14   xmlns:owl="&owl;#"
15   xmlns:service="&service;#"
16   xmlns:profile="&profile;#"
17   xmlns:process="&process;#"
18   xmlns:grounding="&grounding;#"
19   xmlns:tradingServiceProfile=&profile;#
20   xmlns:tradingServiceModel=&process;#
21   xmlns:tradingServiceGrounding=&grounding;#
22   xmlns="&DEFAULT;#"
23   xml:base="&DEFAULT;" >
24   <owl:Ontology rdf:about="" >
25     <owl:versionInfo >
26       $Id:Service.owl generated at: 31/3/2013 10:16:20
27     </owl:versionInfo >
28     <rdfs:comment >
29       This ontology represents the OWL-S service description for the tradingService web service.
30     </rdfs:comment >
31     <owl:imports rdf:resource="&tradingService_service;" / >
32     <owl:imports rdf:resource="&tradingService_profile;" / >
33     <owl:imports rdf:resource="&tradingService_process;" / >
34     <owl:imports rdf:resource="&tradingService_grounding;" / >
35   </owl:Ontology >
36   <service:Service rdf:ID="tradingServiceService">
37     <!-- Reference to the Profile -->
38     <service:presents rdf:resource="&trade_profile; #ServiceProfile / >
39     <!-- Reference to the Process Model -->
40     <service:describedBy rdf:resource=&exchange_process; #ServiceModel/ >
41     <!-- Reference to the Grounding -->
42     <service:supports rdf:resource= &tradeAccess_grounding; #ServiceGrounding'/ >
43   </service:Service >
44   ...
45 </rdf:RDF>

```

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

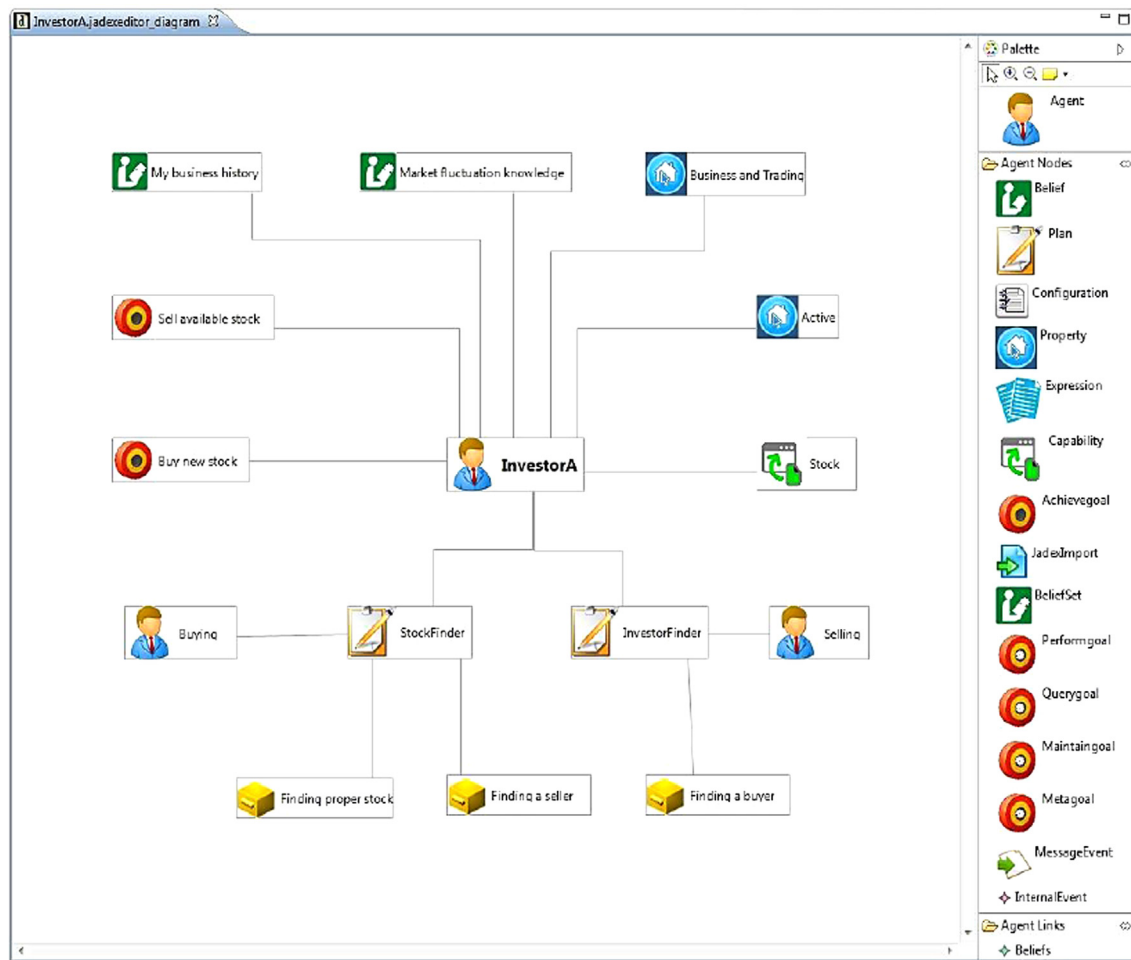


Fig. 15. Partial graphical model of the transformed agent internal viewpoint inside the platform-specific modeling tool (Kardas et al., 2009b) for JADEX agents.

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

AAALADIN (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 called 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.

7.3. A comparative analysis of SEA_ML

Although major differences and/or benefits of SEA_ML with respect to each related work have already been given in the preceding subsections, it is also worth discussing the comparative analysis of SEA_ML in order to both position SEA_ML inside the current big picture of MAS modeling and model-driven MAS development literature and contrast SEA_ML's own approach by making it much more apparent. For this purpose, we have adopted and extended the framework recently proposed in Kardas (2013) for the evaluation of model-driven MAS development studies.

Since the evaluation presented in Kardas (2013) mainly concentrates on the MDD of MAS and does not consider the DSL/DSML perspective, interpretation of some of the original criteria has been altered (e.g. generic PIMM criterion has been evolved into an abstract syntax definition, criterion on the support for multiple platforms has been merged with M2M transformability) and also in this paper new criteria (e.g. concrete syntax definition, viewpoint inclusion) have been added to the evaluation framework for a proper evaluation. As a result, following 6 criteria are used for the comparison of MAS modeling and agent DSL/DSML studies. The interpretation of three-level grading (–, +, ++) for each criterion is given in parenthesis next to the related grade:

1. *Abstract syntax definition*: provision of at least a vocabulary of MAS concepts. A metamodel describing the concepts and their relations is preferred. For this criterion, a study is graded as – (not available), + (abstract syntax is available but specific for a dedicated methodology and/or does not provide a complete metamodel) or ++ (an abstract syntax which enables the development of MASs in various MAS platforms).
2. *Concrete syntax definition*: providing a mapping between agent meta-elements and their representations for models. For this criterion, we grade a related work as – (not available), + (at least textual or a graphical concrete syntax is defined) or ++ (both textual and graphical concrete syntaxes are provided).
3. *Viewpoint inclusion*: the criterion for investigating whether the approach in question supports agent and MAS aspects inside various viewpoints. Agent-internal (AIV), MAS (MASV), Organization (OV) and Interaction (IV) viewpoints, on which the agent research community is mostly agreed (Pavon et al., 2006; Hahn et al., 2009; Beydoun et al., 2009), are taken into consideration as well as the following specific ones: Plan (PV), Role (RV), Behavior (BV) (Hahn, 2008), Goal (GV) (Pavon et al., 2006), Environment (EV) (Hahn, 2008; Omicini et al., 2008; Challenger et al., 2011) and Agent-Service Interaction (ASIV) (Hahn et al., 2008; Omicini et al., 2008; Kardas et al., 2009a; Challenger et al., 2011; Kumar, 2012). For this criterion, a study is marked as (–) if it does not bring an approach apparently based on the viewpoint(s).
4. *M2M transformability*: inclusion of transformable MAS metamodel(s) and a model-to-model transformation mechanism as proposed in Kardas (2013). For this criterion, a study is graded as – (transformation of models is missing), + (a transformation mechanism exists but it is too abstract and not implemented) or ++ (a complete model-to-model transformation phase in which entity mappings between models and the implementation of the written model transformation rules are all included).
5. *M2C transformability*: inclusion of a model-to-code (M2C) transformation phase for automatic generation of MAS software codes. With M2M and M2C transformability criteria, we check whether the approach/the study in question provides some kind of semantics for MAS. For this criterion, a study is graded as – (generation of software codes from MAS models is missing), + (transformations are defined but not implemented or transformations are incomplete and code generation needs too much intervention) or ++ (there is support for an automatic and a complete code generation from models for at least one MAS software development framework and generated template codes are executable).
6. *Tool support*: related work in question is evaluated by considering the support of software modeling and code generation tools for the MAS developers. For this criterion, a study is graded as – (no tool support), + (textual or graphical modeling of MASs is realized only and/or extra mechanism is needed for the support of model transformation) or ++ (various software tools exist for graphical MAS modeling, application of automatic transformation between models and generating MAS software codes from MAS models).

Evaluation results of both the related work and SEA_ML according to the criteria discussed above are given in Table 7. We excluded the studies (except Bauer et al., 2001; Cervenka et al., 2005; Pavon et al., 2006; Kardas et al., 2009a) discussed in Section 7.1 during this evaluation since those studies solely cover metamodel derivation for agent systems and concrete syntax definition in some degree. Hence, we believe that it would be unfair for those studies to be compared with the furnished model-driven MAS development approaches and MAS DSL/DSML proposals.

Comparison results in Table 7 show that SEA_ML and the studies in Hahn (2008), Hahn et al. (2008), Fuentes-Fernandez et al. (2010), Ciobanu and Juravle (2012) and Gascuena et al. (2012) have the expected features of being a complete DSL/DSML for the development of agent systems. Those studies cover the definition of an abstract syntax, a concrete syntax and operational semantics at least, and provide some mechanisms for the exact implementation of MASs. However, some of them do not consider the general purpose (as can be seen from the grades given in the syntax columns of Table 7). For instance, DSL introduced in Ciobanu and Juravle (2012) only provides for the

Table 7

A comparison of model-driven MAS development and agent DSL/DSML studies.

	<i>Abstract syntax definition</i>	<i>Concrete syntax definition</i>	<i>Viewpoint inclusion</i>	<i>M2M transformability</i>	<i>M2C transformability</i>	<i>Tool support</i>
Bauer et al., 2001	++	+	AIV, IV, BV	–	–	+
Cervenka et al., 2005	++	+	AIV, MASV, IV, BV, EV	–	–	+
Kulesza et al., 2005	++	+	MASV, IV, RV, GV	–	+	+
Pavon et al., 2006	++	+	AIV, MASV, IV, GV, EV	–	+	+
Rougemaille et al., 2007	+	–	AIV, MASV, IV	+	–	–
Hahn, 2008 ^a	++	+	AIV, MASV, OV, IV, RV, BV, EV	++	+	++
Hahn et al., 2008	++	+	AIV, MASV, OV, IV, RV, BV, EV, ASIV	+	+	++
Kardas et al., 2009a	++	–	–	++	+	++
Fuentes-Fernandez et al., 2010	+	+	AIV, IV, GV	–	+	+
Ciobanu and Juravle, 2012	+	+	–	–	++	+
Gascuena et al., 2012	+	++	AIV, RV, PV, GV	++	+	++
SEA_ML	++	++	AIV, MASV, IV, PV, RV, EV, ASIV	++	++	++

^a This work was graded by taking into consideration other supporting works (Warwas and Hahn, 2008; Hahn and Fischer, 2009; Hahn et al., 2009) of the same research group in order to cover all features of the proposed MAS DSML.

development of mobile agents while Gascuena et al. (2012) introduces a model just within the scope of Prometheus methodology. On the other hand, SEA_ML and DSML4MAS (Hahn, 2008) differentiate from the remaining studies by introducing general metamodels with various viewpoints that enable the development of MAS for many application domains. Further, as discussed previously, SEA_ML supports the interaction of agents with semantic web services and introduces new viewpoints for agent internals when considering semantic web support. Although it can be said that Hahn et al. (2008) also share the same service interaction perspective, SEA_ML differentiates from their approach by the provision of a unique metamodel as previously discussed during the comparison of SEA_ML and Hahn et al. (2008) in Section 7.2 of this paper. Finally, as far as we know, SEA_ML is the first domain-specific language which provides both textual (Demirkol et al., 2012) and graphical concrete syntaxes for MAS developers. This is why the SEA_ML is marked as the only DSL which receives the full grade for concrete syntax definition in Table 7.

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 approximately 800 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

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://www.mas.ube.edu.tr/downloads/sea_ml_bundle.zip. The bundle also includes more case studies demonstrating SEA_ML use in different application domains.

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 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.

Acknowledgments

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 wish 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, A., Karsai, G., Neema, S., Shi, F., Vizhanyo, A., 2006. The design of a language for model transformation. *Softw. Syst. Model.* 5 (3), 261–288.
- AOS, Agent Oriented Software Pty., Ltd., 2001. JACK Environment. Available from: <http://www.aosgrp.com/products/jack/> (accessed November 2013).
- ATLAS Group, LINA & INRIA, 2006. ATL User Manual. Available from: [http://www.eclipse.org/m2m/atl/doc/ATL_User_Manual\[v0.7\].pdf](http://www.eclipse.org/m2m/atl/doc/ATL_User_Manual[v0.7].pdf) (accessed November 2013).
- Badica, C., Budimac, Z., Burkhard, H., Ivanovic, M., 2011. Software Agents: languages, tools, platforms. *Comput. Sci. Inf. Syst.* 8 (2), 255–298.
- Badica, C., Ilie, S., Kamermans, M., Pavlin, G., Penders, A., 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 Sci. Peace Secur. Ser. D Inf. Commun. Secur.* 32 (12), 286–306.
- Bauer, B., Muller, J.P., Odell, J., 2001. Agent UML: a formalism for specifying multiagent software systems. *Int. J. Softw. Eng. Knowl. Eng.* 11 (3), 207–230.
- Bauer, B., Odell, J., 2005. UML 2.0 and agents: how to build agent-based systems with the new UML standard. *Eng. Appl. Artif. Intell.* 18, 141–157.
- Berners-Lee, T., Hendler, J., Lassila, O., 2001. The semantic web. *Sci. Am.* 284 (5), 34–43.
- Bernon, C., Gleizes, M.-P., Peyruqueou, S., Picard, G., 2003. ADELFE: a methodology for adaptive multi-agent systems engineering. *Lect. Notes Artif. Intell.* 2577, 70–81.
- Bernon, C., Cossentino, M., Gleizes, M.-P., Turci, P., Zambonelli, F., 2005. A study of some multi-agent meta-models. *Lect. Notes Comput. Sci.* 3382, 62–77.
- Bellifemine, F., Poggi, A., Rimassa, G., 2001. Developing multi-agent systems with a FIPA-compliant agent framework. *Softw. Pract. Exp.* 31 (2), 103–128.
- Beydoun, G., Low, G.C., Henderson-Sellers, B., Mouratidis, H., Gomez-Sanz, J.J., Pavon, J., Gonzalez-Perez, C., 2009. FAML: a generic metamodel for MAS development. *IEEE Trans. Softw. Eng.* 35 (6), 841–863.
- 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. *Comput. Sci. Inf. Syst.* 8 (2), 225–253.
- Cervinka, R., Trencansky, I., Calisti, M., Greenwood, D., 2005. AML: Agent Modeling Language—toward industry-grade agent-based modeling. *Lect. Notes Comput. Sci.* 3382, 31–46.
- Challenger, M., Getir, S., Demirkol, S., Kardas, G., 2011. A domain specific metamodel for semantic web enabled multi-agent systems. *Lect. Notes Bus. Inf. Process.* 83, 177–186.
- Ciobanu, G., Juravle, C., 2012. Flexible software architecture and language for mobile agents. *Concurr. Comput. Pract. Exp.* 24 (6), 559–571.
- Clark, T., Evans, A., Sammut, P., Willans, J., 2004. Language driven development and MDA. *MDA J.* 10, 2–13.
- Cossentino, M., 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.
- Demirkol, S., Challenger, M., Getir, S., Kosar, T., Kardas, G., Mernik, M., 2012. SEA_L: a domain-specific language for semantic web enabled multi-agent systems. In: *Proceedings of the 2nd Workshop on Model Driven Approaches in System Development (MDASD 2012)*, held in conjunction with 2012 Federated Conference on Computer Science and Information Systems (FedCSIS 2012), Wroclaw, Poland, IEEE Conference Publications, pp. 1373–1380.
- Depke, R., Heckel, R., Kuster, J.M., 2001. Agent-oriented modeling with graph transformations. *Lect. Notes Comput. Sci.* 1957, 105–119.
- van Deursen, A., Klint, P., Visser, J., 2000. Domain-specific languages: an annotated bibliography. *ACM SIGPLAN Notices* 35 (6), 26–36.
- Duddy, K., Gerber, A., Lawley, M., Raymond, K., 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.
- El-Sawi, K., Ali, D., 1998. Mobile agent technology for dynamic stock exchange. In: *Proceedings of International Conference on Industry, Engineering, and Management Systems (IEMS '98)*, Cocoa Beach, FL, USA.
- Ferber, J., 1999. *Multi-agent Systems: An Introduction to Distributed Artificial Intelligence*. Addison-Wesley Professional, Boston, MA, USA p. 528.
- Ferber, J., 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, T., Labrou, Y., Mayfield, J., 1997. KQML as an agent communication language. In: Bradshaw (Ed.), *Software Agents*. AAAI Press/MIT Press, Cambridge, pp. 291–316.
- Foundation for Intelligent Physical Agents (FIPA), 2002a. FIPA Agent Communication Language Message Structure Specification. Available from: <http://www.fipa.org/specs/fipa00061/> (accessed November 2013).
- Foundation for Intelligent Physical Agents (FIPA), 2002b. FIPA Agent Communication Language Specifications. Available from: <http://www.fipa.org/repository/aclspecs.html> (accessed November 2013).
- IEEE Foundation for Intelligent Physical Agents (FIPA) Modeling Technical Committee (Modeling TC), 2003. Available from: <http://www.fipa.org/activities/modeling.html> (accessed November 2013).
- Fowler, M., 2011. *Domain-Specific Languages*. Addison-Wesley Professional, Boston, MA, USA p. 640.
- Freemind, 2002. Available from: http://freemind.sourceforge.net/wiki/index.php/Main_Page (accessed November 2013).
- 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. *Eng. Appl. Artif. Intell.* 23 (3), 432–444.
- Gascuena, J.M., Navarro, E., Fernandez-Caballero, A., 2012. Model-driven engineering techniques for the development of multi-agent systems. *Eng. Appl. Artif. Intell.* 25 (1), 159–173.

- Getir, S., Demirkol, S., Challenger, M., 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.
- Gray, J., Tolvanen, J.-P., Kelly, S., Gokhale, A., Neema, S., Sprinkle, J., 2007. Domain-specific modeling. In: Fishwick (Ed.), *Handbook of Dynamic System Modeling*. CRC Press, Boca Raton, FL, USA, pp. 1–7.
- Haag, S., Cummings, M., McCubrey, D.J., 2004. *Management Information Systems for the Information Age*, 4th ed. McGraw Hill, Boston, MA, USA.
- 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, C., Fischer, K., 2009. The formal semantics of the domain specific modeling language for multi-agent systems. *Lect. Notes Comput. Sci.* 5386, 145–158.
- Hahn, C., Nesbitt, S., Warwas, S., Zinnikus, I., Fischer, K., 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, C., Madrigal-Mora, C., Fischer, K., 2009. A platform-independent metamodel for multiagent systems. *Auton. Agents Multi-agent Syst.* 18 (2), 239–266.
- Howden, N., Ronnquist, R., Hodgson, A., 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, 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.
- Istanbul Stock Exchange. (<http://www.ise.org>) (accessed November 2013).
- JADEX BDI Agent System, 2003. Available from: (<http://jadex-agents.informatik.uni-hamburg.de/xwiki/bin/view/About/Overview>) (accessed November 2013).
- Apache Jena, 2011. Available from: (<http://jena.apache.org/>) (accessed November 2013).
- Jouault, F., Kurtev, I., 2006. Transforming Models with ATL. *Lect. Notes Comput. Sci.* 3844, 128–138.
- Jouault, F., Allilaire, F., Bezivin, J., Kurtev, I., 2008. ATL: a model transformation tool. *Sci. Comput. Program.* 72 (1–2), 31–39.
- Kalins, A., Barzdins, J., Celms, E., 2005. Model transformation language MOLA. *Lect. Notes Comput. Sci.* 3599, 62–76.
- Kardas, G., 2013. Model-driven development of multi-agent systems: a survey and evaluation. *Knowl. Eng. Rev.* 28 (4), 479–503, <http://dx.doi.org/10.1017/S0269888913000088>.
- Kardas, G., Goknil, A., Dikenelli, O., Topaloglu, N.Y., 2009a. Model driven development of semantic web enabled multi-agent systems. *Int. J. Coop. Inf. Syst.* 18 (2), 261–308.
- Kardas, G., Ekinci, E.E., Afsar, B., Dikenelli, O., Topaloglu, N.Y., 2009b. Modeling tools for platform specific design of multi-agent systems. *Lect. Notes Artif. Intell.* 5774, 202–207.
- Kardas, G., Demirezen, Z., 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.
- Kardas, G., Challenger, M., Yildirim, S., Yamuc, A., 2012. Design and implementation of a multi-agent stock trading system. *Softw. Pract. Exp.* 42 (10), 1247–1273.
- Kelly, S., Tolvanen, J.-P., 2008. *Domain-Specific Modeling: Enabling Full Code Generation*. John Wiley & Sons, Inc., New Jersey, USA.
- Kendall, G., Su, Y., 2003. The co-evolution of trading strategies in a multi-agent based simulated stock market through the integration of individual learning and social learning. In: Proceedings of the International Conference on Machine Learning and Applications (ICMLA'03), Los Angeles, USA, pp. 200–206.
- Kos, T., Kosar, T., Knez, J., Mernik, M., 2011. From DCOM interfaces to domain-specific modeling language: a case study on the Sequencer. *Comput. Sci. Inf. Syst.* 8 (2), 361–378.
- Kosar, T., Carver, J., Mernik, M., 2012. Program comprehension of domain-specific and general-purpose languages: comparison using a family of experiments. *Empir. Softw. Eng.* 17 (3), 276–304.
- Kulesza, U., Garcia, A., Lucena, C., Alencar, P., 2005. A generative approach for multi-agent system development. *Lect. Notes Comput. Sci.* 3390, 52–69.
- Kumar, S., 2012. *Agent-Based Semantic Web Service Composition*. Springer Briefs in Electrical and Computer Engineering. Springer, New York, Heidelberg, Dordrecht, London, p. 57.
- 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 from: (<http://www.isis.vanderbilt.edu/Projects/gme/>) (accessed November 2013).
- Luo, Y., Liu, K., Davis, D.N., 2002. A multi-agent decision support system for stock trading. *IEEE Netw.* 16 (1), 20–27.
- 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 from: (<http://www.w3.org/Submission/OWL-S/>) (accessed November 2013).
- Mernik, M. (Ed.), 2013. *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments*. IGI Global, Hershey, PA, USA.
- Mernik, M., Heering, J., Sloane, A., 2005. When and how to develop domain-specific languages. *ACM Comput. Surv.* 37 (4), 316–344.
- MetaCase, MetaEdit+ Domain-Specific Modeling (DSM) environment, 1995. Available from: (<http://www.metacase.com/products.html>) (accessed November 2013).
- Microsoft DSL Tools, 2005. Available from: (<http://www.microsoft.com/en-us/download/details.aspx?id=2379>) (accessed November 2013).
- Molesini, A., Denti, E., Omicini, A., 2005. MAS meta-models on test: UML vs. OPM in the SODA case study. *Lect. Notes Artif. Intell.* 3690, 163–172.
- Odell, J., Nodine, M., Levy, R., 2005. A metamodel for agents, roles and groups. *Lect. Notes Comput. Sci.* 3382, 78–92.
- Oldevik, J., Neple, T., Gronmo, R., Aagedal, J., Berre, A.J., 2005. Toward standardised model to text transformations. *Lect. Notes Comput. Sci.* 3748, 239–253.
- Object Management Group, Unified Modeling Language Specification, 2000. Available from: (<http://www.uml.org/>) (accessed November 2013).
- Object Management Group, Meta Object Facility (MOF), 2002. Available from: (<http://www.omg.org/spec/MOF/>) (accessed November 2013).
- Object Management Group, Model Driven Architecture (MDA) Specification, 2003. Available from: (<http://www.omg.org/mda/>) (accessed November 2013).
- Object Management Group, Software Process Engineering Metamodel Specification Version 2.0, formal/2008-04-01, 2008. Available from: (<http://www.omg.org/spec/SPM/2.0/>) (accessed November 2013).
- Object Management Group, Ontology Definition Metamodel (ODM), 2009. Available from: (<http://www.omg.org/spec/ODM/1.0/>) (accessed November 2013).
- Object Management Group, Object Constraint Language (OCL), 2012. Available from: (<http://www.omg.org/spec/OCL/2.3.1/>) (accessed November 2013).
- Omicini, A., 2000. SODA: societies and infrastructures in the analysis and design of agent-based systems. *Lect. Notes Comput. Sci.* 1957, 185–193.
- Omicini, A., Ricci, A., Viroli, M., 2008. Artifacts in the A&A meta-model for multi-agent systems. *Auton. Agents Multi-agent Syst.* 17 (3), 432–456.
- Padgham, L., Winikoff, M., 2004. *Developing Intelligent Agent Systems: A Practical Guide*. John Wiley and Sons, Chichester, West Sussex, England p. 230.
- 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, Hershey, PA, USA, pp. 236–276.
- Pavon, J., Gomez-Sanz, J.J., Fuentes, R., 2006. Model driven development of multi-agent systems. *Lect. Notes Comput. Sci.* 4066, 284–298.
- 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, New York, NY, USA, pp. 149–174.
- 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, Chichester, UK, pp. 254–258.
- Posada, M., 2006. Strategic software agents in continuous double auction under dynamic environments. *Lect. Notes Comput. Sci.* 4224, 1223–1233.
- Protégé: Ontology Editor and Knowledge Acquisition System, 2004. Available from: (<http://protege.stanford.edu/>) (accessed November 2013).
- Rahimi, S., Tatikunta, R., Ahmad, R., Gupta, B., 2009. A multi-agent framework for stock trading. *Int. J. Intell. Inf. Database Syst.* 3 (2), 203–227.
- Rao, A., 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, S., Migeon, F., Maurel, C., Gleizes, M.-P., 2007. Model driven engineering for designing adaptive multi-agent systems. *Lect. Notes Artif. Intell.* 4995, 318–332.
- Sánchez Cuadrado, J., García Molina, J., 2007. Building domain-specific languages for model-driven development. *IEEE Softw.* 24 (5), 48–56.
- Schmidt, D.C., 2006. Guest editor's introduction: model-driven engineering. *IEEE Comput.* 39 (2), 25–31.
- Shadbolt, N., Hall, W., Berners-Lee, T., 2006. The semantic web revisited. *IEEE Intell. Syst.* 21 (3), 96–101.
- Sprinkle, J., Mernik, M., Tolvanen, J.-P., Spinellis, D., 2009. Guest editors' introduction: what kinds of nails need a domain-specific hammer?. *IEEE Softw.*, 26, pp. 15–18.
- Smith, R.G., 1980. The contract net protocol: high-level communication and control in a distributed problem solver. *IEEE Trans. Comput.* 29 (12), 1104–1113.
- Strembeck, M., Zdun, U., 2009. An approach for the systematic development of domain-specific languages. *Softw. Pract. Exp.* 39 (15), 1253–1292.
- Sycara, K., Paolucci, M., Ankolekar, A., Srinivasan, N., 2003. Automated discovery, interaction and composition of semantic web services. *J. Web Semant.* 1 (1), 27–46.
- The Eclipse Foundation, MOFScript model to text transformation language and tool, 2005. Available from: (<http://www.eclipse.org/gmt/mofscript/>) (accessed November 2013).
- The Eclipse Foundation, Graphical Modeling Framework (GMF), 2006. Available from: (<http://www.eclipse.org/modeling/gmf/>) (accessed November 2013).
- The Eclipse Foundation, ATL Model Transformation Language and Toolkit, 2007a. Available from: (<http://www.eclipse.org/atll/>) (accessed November 2013).
- The Eclipse Foundation, EMFT JET Editor, 2007b. Available from: (<http://www.eclipse.org/modeling/m2t/?project=jet#jet>) (accessed November 2013).

- Varanda-Pereira, M.J., Mernik, M., Da-Cruz, D., Henriques, P.R., 2008. Program comprehension for domain-specific languages. *Comput. Sci. Inf. Syst.* 5 (2), 1–17.
- Vidal, M., Buhler, P.A., Huhns, M.N., 2001. Inside an agent. *IEEE Internet Comput.* 5 (1), 82–86.
- World Wide Web Consortium, Resource Description Framework, 1999. Available from: <http://www.w3.org/RDF/> (accessed November 2013).
- World Wide Web Consortium, OWL Web Ontology Language, 2004. Available from: <http://www.w3.org/TR/owl-features/> (accessed: November 2013).
- Warmer, J., Kleppe, A., 2003. *Object Constraint Language: The Getting Your Models Ready for MDA*, 2nd Edition Pearson Education, USA p. 240.
- Warwas, S., 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.
- Wooldrige, M., Jennings, N.R., 1995. Intelligent agents: theory and practice. *Knowl. Eng. Rev.* 10 (2), 115–152.
- WSDL-S: Web Service Semantics, 2005. Available from: <http://www.w3.org/Submission/WSDL-S/> (accessed November 2013).
- WSMO: Web Service Modeling Ontology, 2005. Available from: <http://www.w3.org/Submission/WSMO/> (accessed November 2013).
- Zambonelli, F., Jennings, N.R., Wooldrige, M., 2003. Developing multiagent systems: the Gaia methodology. *ACM Trans. Softw. Eng. Methodol.* 12 (3), 317–370.