DOI:10.2298/CSIS121105044D

A DSL for the Development of Software Agents working within a Semantic Web Environment

Sebla Demirkol[1], Moharram Challenger[1], Sinem Getir[1], Tomaž Kosar[2], Geylani Kardas[1*], and Marjan Mernik[2]

[1] International Computer Institute, Ege University, Bornova, 35100 Izmir, Turkey
sebla.demirkol@ege.edu.tr, moharram.challenger@mail.ege.edu.tr, sinem.getir@ege.edu.tr, geylani.kardas@ege.edu.tr
[2] Faculty of Electrical Engineering and Computer Science, University of Maribor, Smetanova 17, 2000 Maribor, Slovenia
tomaz.kosar@uni-mb.si, marjan.mernik@uni-mb.si

Abstract. Software agents became popular in the development of complex software systems, especially those requiring autonomous and proactive behavior. Agents interact with each other within a Multi-agent System (MAS), in order to perform certain defined tasks in a collaborative and/or selfish manner. However, the autonomous, proactive and interactive structure of MAS causes difficulties when developing such software systems. It is within this context, that the use of a Domain-specific Language (DSL) may support easier and quicker MAS development methodology. The impact of such DSL usage could be clearer when considering the development of MASs, especially those working on new challenging environments like the Semantic Web. Hence, this paper introduces a new DSL for Semantic Web enabled MASs. This new DSL is called Semantic web Enabled Agent Language (SEA_L). Both the SEA_L user-aspects and the way of implementing SEA_L are discussed in the paper. The practical use of SEA_L is also demonstrated using a case study which considers the modeling of a multi-agent based e-barter system. When considering the language implementation, we first discuss the syntax of SEA_L and we show how the specifications of SEA_L can be utilized during the code generation of real MAS implementations. The syntax of SEA_L is supported by textual modeling toolkits developed with Xtext. Code generation for the instance models are supplied with the Xpand tool.

Keywords: Domain-specific Language, Metamodel, Multi-agent System, Semantic Web.

[*] Corresponding author. Tel:+90-232-3113223 Fax: +90-232-3887230

1. Introduction

Software agents [1] [2] are autonomous software components which are able to act on behalf of their users in order to perform a group of defined tasks. Many intelligent software agents interact with each other within a system called Multi-agent System (MAS). Their interactions can be either cooperative or selfish [45]. Software agents and MASs are recognized as both useful abstractions and effective technologies for the modeling and building of complex distributed systems. The implementation of these autonomous, responsive, and proactive systems is naturally a complex task.

Additionally, the Semantic Web improves the World Wide Web such that the web pages' contents can be interpreted using ontologies [46]. Therefore, this new-generation web helps machines to understand web content. It is apparent that the interpretation in question will be realized by autonomous computational entities (i.e. agents) in order to handle the semantic content on behalf of their users. Surely, a Semantic Web environment has specific architectural entities, and thus a different semantics needs to be considered for modeling a MAS within its environment. Thus, the Semantic Web evolution has spawned a new vision regarding agent research. Software agents are planned for collecting Web content from diverse sources, processing the information, and exchanging the results. 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. However, considering agent interactions with Semantic Web elements adds more complexity for designing and implementing these systems.

On the other hand, the Model Driven Development (MDD) is also one of the important software development approaches, moving software development from code to models [43], which increases productivity [26] and reduces development costs [47]. The design and implementation of a MAS may become more complex when new requirements and interactions for new agent environments like Semantic Web are considered. MDD can provide an infrastructure that simplifies the development of such MASs. Being able to work at a higher abstraction level is of critical importance for the development of MASs since it is almost impossible to observe the code level details of the MASs due to their internal complexity, distributedness and openness. Hence, such an MDD application can increase the abstraction level during MAS development. MDD uses different approaches for realizing its goals. One of these methods is Domain-specific Language (DSL) development [8, 14, 29, 32, 48]. DSLs are languages which are comprised of a domain's concepts and terminologies in order to supply the requirements of the domain. A DSL allows end-user programmers (domain experts) to describe the essence of a problem using abstractions related to a domain specific problem space.

We present a new DSL for designing and implementing MASs working within a Semantic Web environment, by motivating from the expressive powers of DSLs and MDD. We call this new DSL as Semantic web Enabled

Agent Language (SEA_L). An abstract syntax and a concrete syntax for SEA_L are discussed in the paper, that originated from the domain-specific metamodel, which is first introduced in [4]. Furthermore, transformations required for code generation from the specifications of SEA_L are defined in order to realize the implementation of modeled MAS in various agent execution platforms.

This paper is an extended version of the paper [6]. It differs from the latter by including a discussion of all viewpoints, the full specification of two crucial viewpoints of the proposed DSL, and a detailed discussion regarding the practical usage of the language within the scope of a case study. The case study covers the design and real implementation of an agent-based e-barter system. Again different from the paper [6], discussion of the agent-based e-barter business domain is elaborated as well as modeling and code generation for agent internals have been added in this paper. Moreover, in this paper the user and implementation aspects of the proposed DSL are discussed separately. Firstly, we present an overview of the SEA_L language, together with a case study. Then the implementation details are stated.

The remainder of the paper is organized as follows: An overview of the new language is given in Section 2 along with an example. The abstract syntax, the textual concrete syntax, and the code generation mechanism for new DSL are discussed in Section 3. In Section 4, the related work is presented. Finally, Section 5 concludes the paper, and states the future work.

2. The SEA_L Domain-Specific Language

In order to separate the 'user' aspects of the SEA_L from its implementation details, in this section we present SEA_L concepts and how to use them, along with a case study and in the next section a discussion on the implementation details of SEA_L.

Since SEA_L is designed for developers of MASs working within the Semantic Web environments, the language's main concepts consist of both MAS and Semantic Web terminologies.

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 a content language.

SEA_L is divided into eight viewpoints in order to provide clear understanding and efficient usage. These viewpoints are:

1. Agent Internal Viewpoint: This viewpoint is related to the internal structures of semantic web agents (SWA) and defines those entities and their relations required for the construction of agents. It covers both reactive and Belief-Desire-Intention (BDI) [41] agent architectures.

2. Interaction Viewpoint: This aspect of the language expresses the interactions and communications in a MAS by taking messages and message sequences into account.
3. MAS Viewpoint: This viewpoint solely deals with the construction of a MAS as a whole. It includes those main blocks of which the complex system is composed as an organization.
4. Role Viewpoint: This perspective delves into the complex controlling structure of the agents. All role types such as Ontology Mediator Role or Registration Role are modeled in this viewpoint.
5. Environmental Viewpoint: Agents may need to access some resources (e.g. services and knowledge-bases (considering the facts about the surroundings)) within their environments. The usage of resources and the interactions of agents with their surroundings are covered in this viewpoint.
6. Plan Viewpoint: This viewpoint deals particularly with Plans' internal structures. Plans are composed of some Tasks and atomic elements such as Actions.
7. Ontology Viewpoint: SWAs know various ontologies as they work with Semantic Web Services (SWS) and also some ontological concepts which constitute agents' knowledge-bases (such as belief and fact).
8. Agent-SWS Interaction Viewpoint: This is probably the most important viewpoint of SEA_L. The interactions of agents with SWSs are described within this viewpoint. Entities and relations are defined for service discovery, agreement, and execution. The internal structures of SWSs are also modeled.

SemanticWebAgent (SWA) in SEA_L stands for each agent within the Semantic Web enabled MAS. A SemanticWebAgent is an autonomous entity which is capable of interaction with both other agents and *SemanticWebService*s (*SWS*) within the environment. SemanticWebAgents can be associated with more than one *Role* at any time (multiple classifications), and can change roles over time (dynamic classification). An agent can play roles within various environments, have a state (*Agent State*), and own a type (*Agent Type*) during its execution.

A SemanticWebAgent can interact with various services including SemanticWebServices. A SemanticWebService represents a service (except for an agent service), its capabilities, and its interactions, semantically. A SemanticWebService is composed of one or more *Web Service* entities. The corresponding services must have a semantic interface that is going to be used by platforms' agents.

A SemanticWebAgent applies Plans to perform their Tasks. 'Semantic Service Register Plan' (SS_RegisterPlan), 'Semantic Service Finder Plan' (SS_FinderPlan), 'Semantic Service Agreement Plan' (SS_AgreementPlan) and 'Semantic Service Executor Plan' (SS_ExecutorPlan) are extensions of the Plan. Agents use the SS_RegisterPlan for communication with a service registry to discover service capabilities. Other Plans are used to discover SemanticWebServices dynamically, call the services, obtain agreement with

them and execute them, respectively. Finally, a SSMatchmakerAgent can play a RegistrationRole to advertise a SemanticWebService.

SEA_L also covers the already expected and traditional MAS entities (in addition to above mentioned items) such as *Capabilities*, *Goal*, *Belief,* and so on. SEA_L also defines various relations for these entities such as *appliesPlan*, *includesBelief*, *usesGoal*, *postCondition*, *realized_by,* and so on. When considering SWSs and their use within MASs, there are entities like *Grounding*, *Process*, *Interface*, *SSMatchmakerAgent*, *RegistrationRole*, and different types of plans. When taking into account the relations regarding agents and SWS interactions, SEA_L contains relations like *appliesPlan*, *playsRole*, *executes*, *uses*, *interactsWith*, *describes*, *presents*, and *supports*. Using these relations, a developer can model a high-level program for MASs working within Semantic Web environments.

2.1. Case Study: E-Barter System

SEA_L can be used in many instances for facilitating the design and development of agent-based systems for various domains such as agent-based business evaluation [30], stock exchange [24], document management [40] and the e-barter system [7]. In order to exhibit the use of the introduced DSL, the modeling of a simple multi-agent based e-barter system is considered during this study. A barter system is an alternative commercial approach where customers meet at a marketplace in order to exchange their goods or services without currency. In barter marketplaces, purchased goods or services are exchanged for manufactured goods or offered services [7].

An agent-based e-barter system consists of agents that the exchange goods or services of owners according to their preferences. In this application, the base scenario is achieved by the *Customer*, '*Barter Manager*' and *Cargo* agents. Interested readers may refer to [7] for a detailed discussion of barter proposals and the tracking of the bargaining process between Customer agents. After the finalization of bargaining, Customer agents send engagement message to the '*Barter Manager*' agent. The '*Barter Manager*' agent notifies the Cargo agent for transporting barter products between Customer agents. This scenario is completed by the acceptances of all participating agents.

For instance, two Customer Agents (one from the automotive industry and another from the healthcare sector) may need to exchange their offered goods and services such that: the car manufacturer offers to sell car spare-parts to a health insurance company (e.g., for the health company's service cars), and wants to procure health insurance for its employees. Let us consider that the intention of the health insurance company is vice-versa. During bargaining between the agents of the car manufacturer and the health insurance company, our Barter Manager agent uses a semantic web service called '*Barter Service*'. In order to invoke this service, the '*Barter Manager*' first needs to discover the proper semantic web service. Then, it interacts

with the candidate service(s) and after an agreement the exact execution of the semantic web service is realized [25]. Figure 1 portrays the partial instance model of an E-Barter system (conforming to the SEA_L's metamodel, as elaborated in Section 3.1).

In the following, we provide a description of the instances and constraint controls for this case study using SEA_L specifications.

Listing 1 shows the textual instance model for the Agent Internal viewpoint of the E-barter system. The instance model includes those variables and relations defined for the E-barter domain. Also, according to the syntax of SEA_L's Agent Internal viewpoint (which is discussed in subsection 3.2), there should be at least one instance of SemanticWebAgent and Capabilities within the system. Therefore, initially, a SemanticWebAgent and Capabilities have been defined for this example.

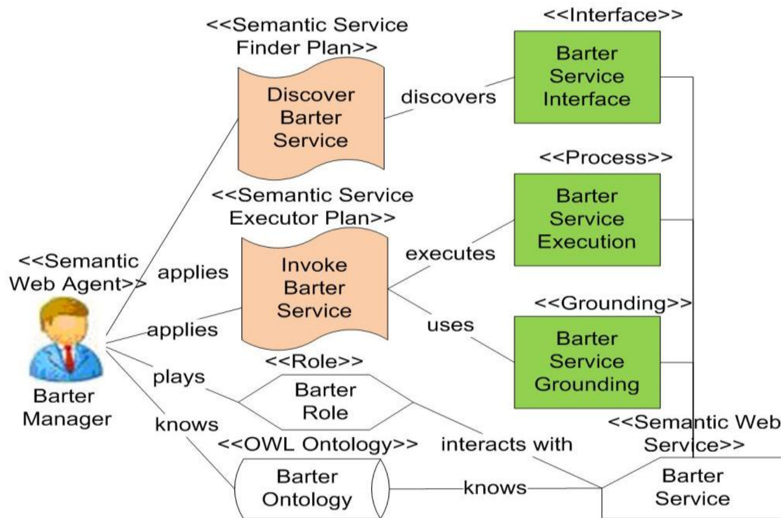

Figure 1. Overview of the E-Barter system as a SEA_L instance [25]

```
01      AgentInternalViewPoint e_barter {
02          SemanticWebAgent barterManager
03              "Barter Manager Agent"      // Agent Description
04              "Properties"                // Agent Properties
05              "Active"                    // Agent State
06              "CustomerAgent";            // Agent Type
07          Capabilities barterCap
08              "Barter Manager Capability";
09          Role barterRole;
10          Goal bestMatching
11              "Doing best matching" 1;     // Recur = 1
12          Belief barterKnowledge
13              "System facts" 2;           // Dynamic = 2
14          Plan financialPlan
15              "Cyclic Plan" 1;            // Priority = 1
16          barterManager{
17              includes barterCap;
18              plays barterRole;
19          }
20          barterCap{
21              appliesPlan financialPlan;
22              includesBelief barterKnowledge;
23              usesGoal bestMatching;
24          }
25          barterKnowledge{ precondition bestMatching; }
26          bestMatching{
27              postcondition barterKnowledge;
28              realized_by financialPlan;
29          }
30      }
```

Listing 1. Textual modeling for Agent Internal viewpoint of a multi-agent e-barter system in SEA_L

Listing 2 shows the use of SEA_L in textual modeling of Agent-SWS Interaction viewpoint of the multi-agent e-barter system in question. In order to infer about the semantic closeness between offered and purchased items based on the defined ontologies, a SemanticWebAgent is defined which can use a SemanticWebService called barterService.

```
01  SWSInteractionViewPoint  e_barter_Interaction{
02      SemanticWebAgent barterManager
03          "Barter Manager Agent"       // Agent Description
04          "Properties";                 // Agent Properties
05      SWS barterService;
06      SSMatchmakerAgent barterMatchAgent
07          "E-Barter Matchmaker Agent"
08          "Properties";
09      Grounding barterServiceGrounding;
10      Process barterServiceProcess;
11      Interface barterServiceInterface;
12      SS_RegisterPlan serviceRegistration;
13      SS_FinderPlan discoverBarterService;
14      SS_AgreementPlan negotiating;
15      SS_ExecutorPlan invokeBarterService;
16      Role barterRole;
17      RegistrationRole matchRole;
18      barterManager{
19          appliesPlan discoverBarterService;
20          appliesPlan negotiating;
21          appliesPlan invokeBarterService;
22          playsRole barterRole;
23      }
24      barterMatchAgent{
25          appliesPlan serviceRegistration;
26          playsRole matchRole;
27      }
28      invokeBarterService{
29          executes barterServiceProcess;
30          uses barterServiceGrounding;
31      }
32      discoverBarterService { interactsWith barterMatchAgent; }
33      barterRole { interactswith barterService; }
34      barterServiceProcess {describes barterService;}
35      barterServiceInterface { presents barterService;}
36      barterServiceGrounding { supports barterService;}
37  }
```

Listing 2. Textual modeling for Agent-SWS Interaction viewpoint of a multi-agent e-barter system in SEA_L

barterManager is an instance of the SemanticWebAgent, which has an important role named barterRole within the system, and applies the discoverBarterService plan, which is an instance of the SS_FinderPlan for finding the desired services. In addition, the agent applies a '*negotiating'* plan, which is an SS_AgreementPlan for negotiating with the discovered services. It also applies the invokeBarterService plan that is an instance of the SS_ExecutorPlan for executing the agreed service. discoverBarterService

discovers the barterServiceInterface which presents a barterService. Moreover, invokeBarterService uses barterServiceGrounding for knowing about the execution protocol of the service, and executes barterServiceProcess which declares the internal process of the service.

barterService is an instance of the SemanticWebService, and is described by the barterServiceProcess. This system also has an SS_Matchmaker Agent called the barterMatchAgent, which applies serviceRegistration as an SS_RegistrationPlan for realizing the registration of Interfaces for SemanticWebServices.

In order to provide more readability for Agent-SWS interaction within the code, defining plans, SS_RegisterPlan, SS_FinderPlan, SS_AgreementPlan and SS_ExecutorPlan must be in order, as shown in Listing 2. Otherwise, the SEA_L editor will indicate an error.

As it is restricted in textual concrete syntax, each instance model must have at least one SemanticWebAgent and one SemanticWebService (see Listing 2). After the declarations, the barterManager, being a SemanticWebAgent, applies the *discoverBarterService* plan for finding candidate services, the '*negotiating'* plan for making an agreement with one of them, and the *invokeBarterService* for executing the agreed service. It also plays a *barterRole* for accomplishing these interactions. The *discoverBarterService* plan interacts with the barterMatchAgent and the Matchmaker Agent, in order to find the candidate services. After this interaction, the result is discovering a set of barterServiceInterfaces.

At the end of the SS_FinderPlan, the SS_ExecutorPlan starts which executes the Process and uses Grounding. Moreover, the Role interacts with the SemanticWebService which is presented by the Interface, describes the Process and is supported by the Grounding. Finally, the SemanticWebService depends on at least one '*Service Ontology'*.

As will be elaborated in subsection 3.3 of this paper, by applying the rules written in Xpand [50], the SEA_L's code generation feature enables agent developers to automatically obtain 1) agent software codes conforming to the JADEX [23] BDI platform which is one of the popular APIs for developing software agents, 2) Ontology files in OWL [36] format, and 3) OWL-S [37] representations of the modeled SWSs. Therefore, after running the code generation of SEA_L for the case study, a JADEX ADF file for the barterManager agent and a plan file for each Plan element are generated. The generated ADF file can be used inside a JADEX platform in order to initialize the designed barterManager agent and this agent then executes the generated Java plan code in order to do its tasks. An excerpt from the generated plan named the *financialPlan* for the Barter Manager agent is given in Listing 3. This given code is automatically generated as a result of applying the generation rules (as discussed in section 3.3). Based on the transformation, the modeled agents' behavior is implemented as a JADEX Plan class that owns the *'body'* method to cover the required codes for the agent tasks.

Part of the generated ADF file is shown in Listing 4. In this file, all of the keywords and their attributes correspond with the related tags. For example,

the required descriptions for agent capabilities (Lines 14-19), plans to be applied (Lines 20-27), beliefs pertaining to the agent (Lines 28 -32), and the goal of the Barter Manager agent (Lines 33-46) modeled in SEA_L can now be included within a JADEX ADF.

With applying code generations of SEA_L, two ADF files, four plan files, four OWL-S files (Service, Service Process, Service Profile, and Service Grounding), and one WSDL file are generated for SEA_L's Agent-SWS Interaction viewpoint. The ADF and plan files are similar to the ones generated for Agent Internal viewpoint. Therefore, only one part of the generated OWL-S file for '*Barter Service*' is given as an example in Listing 5. Lines from 1 to 9 contain boilerplate text inserted directly from a template (as discussed in subsection 3.3). The barterService, barterServiceInterface, barterServiceProcess and barterServiceGrounding names in lines 24, 27, 30 and 33 of Listing 5 are supplied from the declarations in Listing 2. As previously discussed, a '*Barter Manager*' agent needs a '*Barter Service'* SWS during the bargaining process. Hence, the OWL-S documents referred to in Listing 5 for service interface (in Line 26), service process (Line 29), and finally grounding (Line 32) are used by the agent in order to find, process, and finally invoke the required service.

```
01  import java.util.*;
02  import jadex.runtime.*;
03  import java.util.StringTokenizer;
04  public class financialPlan extends Plan {
05      // Plan attributes.
06      ...
07      // static block or constructor
08      ...
09      // Constructor code.
10      public financialPlan() {
11              ...
12      }
13      // Plan main code.
14      public void body() {
15              // Send request
16              ...
17              // Wait for reply
18              …
19      }
20  }
```

Listing 3. Generated plan file for financialPlan

```
01 <agent
02    xmlns = "http://jadex.sourceforge.net/jadex"
03    xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
04    xsi:schemaLocation = "http://jadex.sourceforge.net/jadex
05    http://jadex.sourceforge.net/jadex-2.0.xsd"
06    name = "barterManager"
07    description = "Barter Manager Agent"
08    properties = "Properties"
09    package=”jadex.examples.myProjects”
10 >
11    <imports>
12          <import>jadex.adapter.fipa.*</import>
13    </imports>
14    <capabilities>
15       <capability>
16          name = "barterCap" file=""
17          description = "Barter Manager Capability"
18       </capability>
19    </capabilities>
20    <plans>
21         <plan name = "financialPlan"
22           description = "Cyclic Plan"
23           priority="1" />
24        <plan name = "discoverBarterService" />
25        <plan name = "negotiating" />
26        <plan name = "invokeBarterService" />
27    </plans>
28    <beliefs>
29          <belief name="barterKnowledge"
30            description="system facts"
31            dynamic="1" />
32    </beliefs>
33    <goals>
34       <achievegoal name="bestMatching"
35         recur = 1
36         exclude = "when_tried"
37         recalculate = "true" retry="true"
38         exported = "false"
39         posttoall = "false"  recurdelay = "0"
40         randomselection = "false"
41         retrydelay = "0">
42         <creationcondition>
43               <!-- Write Creation Condition -->
44         </creationcondition>
45       </achievegoal>
46    </goals>
47 </agent>
```

Listing 4. Part of generated ADF file from Agent Internal viewpoint of barterManager in E-Barter System case study

```
01  <?xml version="1.0" encoding = 'ISO-8859-1'?>
02  <!DOCTYPE ruidef[
03     <!ENTITY barterService_profile
04         "http://mas.ube.ege.edu.tr/ barterServiceProfile.owl">
05     <!ENTITY barterService_process
06         "http://mas.ube.ege.edu.tr/ barterServiceProcess.owl">
07     <!ENTITY barterService_grounding
08         "http://mas.ube.ege.edu.tr/ barterServiceGrounding.owl">
09  ]>
10  <rdf:RDF xmlns:rdf= "&rdf;#" xmlns:rdfs="&rdfs;#"
11     xmlns:owl = "&owl;#" xmlns:service= "&service;#"
12     …
13     xml:base="&DEFAULT;" >
14     <owl:ontology rdf:about="">
15     <owl:versionInfo>
16           $Id: barterService.owl,v 1.14 2012/10/08 15:27:40 $
17     </owl:versionInfo>
18        <rdfs:comment> "This ontology represents the OWL-S
19           service description for the barterService service example."
20        </rdfs:comment>
21        <owl:imports rdf:resource="&service;" />
22        …
23     </owl:Ontology>
24     <service:Service rdf:ID= "barterService">
25           <!-- Reference to the Profile -->
26           <service:presents rdf:resource="&barterService_profile;
27              #barterServiceInterface"/>
28           <!-- Reference to the Process Model -->
29           <service:describedBy rdf:resource="&barterService_process;
30              #barterServiceProcess"/>
31           <!-- Reference to the Grounding -->
32           <service:supports rdf:resource="&barterService_grounding;
33              #barterServiceGrounding"/>
34     </service:Service>
35     <profile:Profile rdf:about=&
36            "barterService_profile;#barterServiceInterface">
37            <service:presents rdf:resource=#"barterService"/>
38     </profile:Profile>
39     <process:AtomicProcess rdf:about=&
40            "barterService_process;# barterServiceProcess">
41             <service:describedBy rdf:resource=#"barterService"/>
42      </process:AtomicProcess>
43      <grounding:WsdlGrounding rdf:about=&
44            "barterService_grounding;# barterServiceGrounding">
45            <service:supports rdf:resource=#"barterService"/>
46     </grounding:WsdlGrounding>
47   </rdf:RDF>
```

Listing 5. Part of generated OWL-S Service file

3. SEA_L Implementation

In this section, the implementation details of SEA_L language are provided including abstract syntax as a metamodel divided into several viewpoints, its textual concrete syntax, and the required code generation for presenting the operational semantics of the language.

3.1. Abstract Syntax

The abstract syntax of a DSL describes the concepts and their relations without any consideration of meaning. In terms of MDD, the abstract syntax is described by a metamodel that defines what the models should look like.

The Platform Independent Metamodel (PIMM) which represents the abstract syntax of SEA_L is divided according to the eight viewpoints which were previously given in section 2.

We discuss the metamodel over its Agent Internal viewpoint as well as Agent-SWS Interaction viewpoint throughout this paper due to the vital importance of these viewpoints. In addition, critical entities from other viewpoints are already considered during the following discussion. The related viewpoints are shown in Figures 2 and 3, respectively. In these Figures, the elements filled-in with light gray come from other viewpoints which are shown on the top or bottom of the element using '<<' and '>>'. In other words, these elements are common elements amongst the viewpoints, and tailor them to each other.

The Agent Internal viewpoint is related to the internal structures of the semantic web agents and defines the entities and their relations required for the construction of agents. A partial metamodel which represents this viewpoint, is given in Figure 2.

SEA_L's metamodel (hence abstract syntax) supports both reactive and Belief-Desire-Intention (BDI) agent architectures. BDI was first proposed by Bratman [3] and is used within many agent systems. In a BDI architecture, an agent decides about which Goals to achieve and how to achieve them. Beliefs represent the information an agent has about its surroundings, while Desires correspond to the things that an agent would like to have achieved. Intentions, which are the deliberative attitudes of agents, include the agent planning mechanism in order to achieve the goals. Taking concrete BDI agent frameworks (such as JADEX [23] and JACK [21]) into consideration, we propose an entity called *Capabilities* which includes each agent's *Goals*, *Plans* and *Beliefs* about the surroundings.

The Agent-SWS Interaction viewpoint focuses on the internal structure of the *SemanticWebServices* and the interaction of any SemanticWebAgent with SemanticWebServices within a MAS organization. Concepts and their relations for appropriate service discovery, agreement with the selected service, and execution of the service are all defined within this viewpoint. A

partial metamodel of SEA_L which represents this viewpoint is shown in Figure 3.

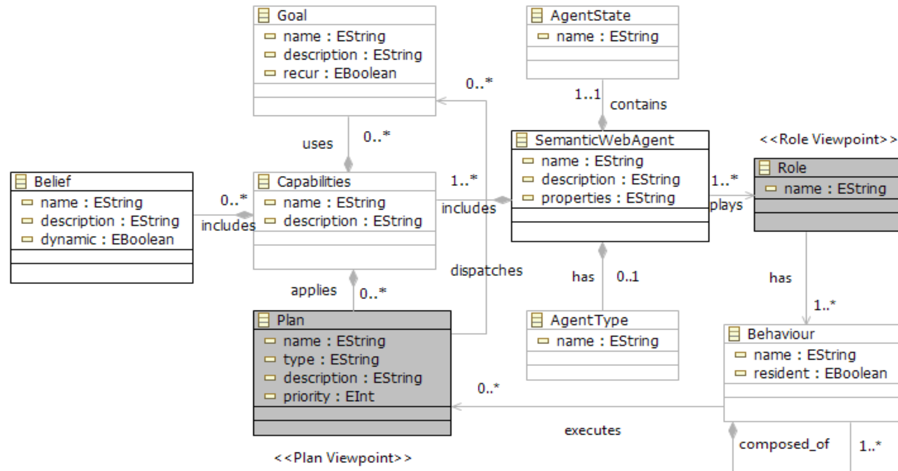

Figure 2. Agent Internal viewpoint

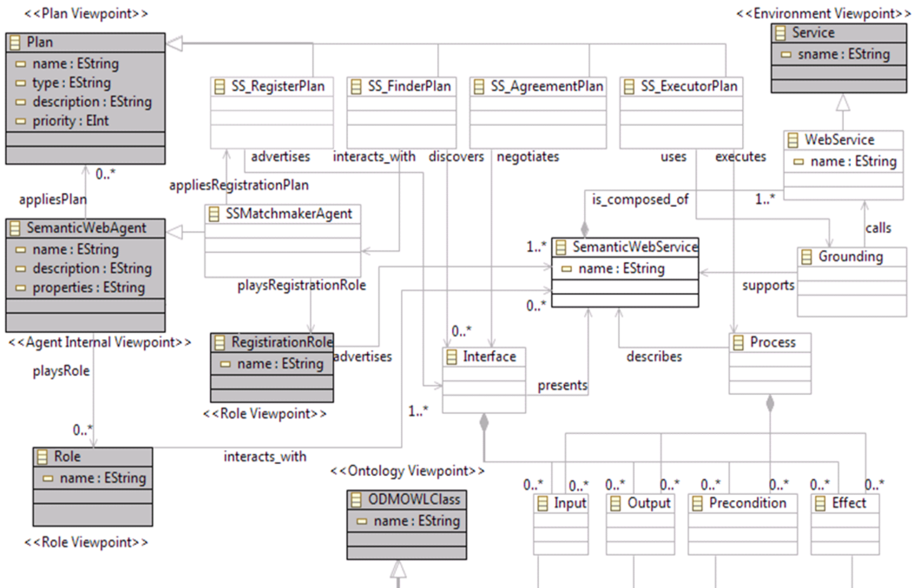

Figure 3. Agent-SWS Interaction viewpoint.

Semantic Web Service (SWS) modeling approaches (i.e. OWL-S [37]) generally define a service with three 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. The '*Process Model*' defines a service's internal combinations and service execution dynamics. Finally, '*Physical Grounding*' defines the service's execution protocol. These meta-entities are shown in Figure 3 with *Interface*, *Process,* and *Grounding* entities, respectively. These components can use *Input*, *Output*, *Precondition,* and *Effect* which are extensions of the Web Ontology Language (OWL [36]) class from Object Management Group's (OMG) Ontology Definition Metamodel (ODM) [35].

On the other hand, agents need to communicate with a service registry element in order to discover service capabilities. Hence, the metamodel includes a specialized agent entity, called the *SSMatchmaker* Agent. This entity represents those matchmaker agents which store the capability advertisements of SemanticWebServices within a MAS, and match those capabilities with service requirements sent by the other platform agents.

When considering the other viewpoints of SEA_L, the MAS viewpoint solely deals with the construction of a MAS as an overall aspect of the metamodel. Plan viewpoint defines a Plan's internal structure. When an Agent applies a Plan, it executes its Tasks. In addition, message transaction is considered within this viewpoint. The Role viewpoint shows distinct types of roles. Agents can use several roles at any time and can alter these roles over time. The Interaction viewpoint focuses on agent communications and interactions in a MAS, and defines entities and relations such as *Interaction*, *Message*, and *MessageSequence*. The Environment viewpoint focuses on the relations between agents and to what they access. Environment contains all non-Agent *Resources*, *Facts,* and *Services*. The Ontology viewpoint brings all ontology sets and ontological concepts together. ODM OWL [36] Ontology from OMG is a standard for all of our ontology sets such as *Role*, *Organization,* and *ServiceOntologies*.

3.2. Textual Concrete Syntax

The textual concrete syntax of SEA_L is provided with Xtext [52]. Xtext is a language development framework for developing textual modeling languages. It can be used for creating a sophisticated Eclipse-based development environment. Xtext is based on EBNF (Extended Backus–Naur Form) [20] rules.

If the metamodel which represents the abstract syntax for SEA_L is considered as an analysis phase of the concrete syntax of SEA_L, the design phase will be the part describing the EBNF rules. One of the main advantages of DSLs is for validating domain-specific constraints. The constraints of the language can be implemented within the '*Validation Package*' in Xtext, which provides a dedicated hook for validation rules. Also, other features of SEA_L's textual concrete syntax are created using both

manually-written code and Xtext features. When using Xtext features, the textual concrete syntax supplies auto completion, syntax coloring, rename refactoring, bracket matching, auto edit, an outline view that shows the semantic structure of the model and code formatting for properly indenting the documents. The above discussed constraints of SEA_L's metamodel, are realized by defining the EBNF rules. With these capabilities, the new DSL possesses both the structural and static semantics of the MAS domain. The structure is defined by the method signatures and the static semantics are defined by the constraint code.

During textual modeling with Xtext, the controls over the instance models can be realized via controlling packages. These packages include formatting, scoping, and validation.

The formatting package (Pretty Printing [12]) simply controls and applies the editorial rules for an instance model. In this package, by accessing the language grammar, we can define additional editorial controls (formatting configuration) in order to modify the written program automatically, which help the instance model to be more readable. For example, spaces for keywords, line-wrap rules, etc can be considered in an instance model of the DSL.

Using the scoping application programming interface (API), it is possible to define which elements are referable by a certain variable reference [12]. In other words, it can be controlled that from which parts of the program, a variable in a scope (a block of code), can be accessed.

One of the interesting aspects of developing a DSL is static analysis or validation of the written program. Validation package plays this role within the Xtext tool. The goal is that the users of the language obtain informative feedback as they type the program [12]. Some of the validations are performed automatically, e.g. syntactical and crosslink validations using parser and linker, respectively; although they can also be customized by the user. This type of validation is done with the help of grammar and scoping. However, in addition to the automatic validations, we can specify additional constraints specific for our Ecore model, called custom validation. For example, it is possible to control the number of specific elements. Although some of the constraints could be fulfilled by grammar terminal rules in Xtext (e.g. controlling the format of the defined variables), we implemented them using the validation package to ease providing desired messages (warning or error), and to provide the possibility for fixing the error or warning. In the remainder of this subsection, we discuss how the textual concrete syntax of SEA_L's major viewpoints is provided with Xtext.

3.3. Textual Concrete Syntax of Agent Internal Viewpoint

An Xtext grammar is structured with rules which are identified by the text to the left of a colon. There is at least one rule for each meta-element within the textual concrete syntax. EBNF rules are defined for Agent Internal viewpoint according to the constraints in the metamodel. The first constraint is that all

of the instance model's elements must be in *AgentInternalViewpoint* tag. Also, the instance model must start and end with curly brackets. An example of another constraint is that each instance model must have at least one SemanticWebAgent and one Capabilities, in any order. These constraints are supplied within AgentInternalViewpoint, rule which is given in Listing 6.
According to Xtext syntax, the assignment operator, '=', denotes a single valued feature, the '+=' operator denotes a multi-valued feature, and the asterisk operator, '*', denotes a cardinality of 0..n. Also, within each rule, referring to predefined parser rules is possible using '[' and ']' characters (called '*cross referencing*'), as shown in Listing 7 Line 3.

```
01  AgentInternalViewpoint:
02    'AgentInternalViewPoint'   '{'
03          semanticwebagent+=SemanticWebAgent &
04          capabilities+=Capabilities
05         …
06    '}';
```

Listing 6. A part of AgentInternalViewpoint rule.

```
01Capabilities:
02  'Capabilities'  name = ID description = STRING';' |
03  cap = [Capabilities]  '{'
04       (  'includes'  belief = [Belief]';' |
05          'uses' goal = [Goal]';' |
06          'applies' plan = [Plan] ';'  )*
07  '}';
```

Listing 7. Capabilities rule

SEA_L's metamodel conforms to BDI [41] architecture. Therefore, a group of meta-elements exists for supplying the BDI structure. When considering this structure, a Capabilities meta-element consists of Belief, Goal, and Plan meta-elements. The user can define numerous relations by considering the Agent Internal viewpoint. This structure is defined within the Capabilities rule, which is shown in Listing 7. The developer can define the Belief, Goal, and Plan meta-elements as often as needed and in any order, regarding lines 4 to 7 of Listing 7.

The agent state and type definitions are considered as string-terminals within the Agent Internal viewpoint, although they could be implemented as hard-coded enumerations or references to their definitions. This is because we believe that agents can conceptually have any user-defined state and type (not limited to specific states or types). Also, in order to have agent definition integration within a single line, we do not use references to agent type and state definitions.

Fewer constraints are defined within the Agent Internal viewpoint in comparison with the Agent-SWS Interaction viewpoint, since the elements

are generally used arbitrarily, and most of the relations are independent within the Agent Internal viewpoint.

The user can assign a keyword to the name of an instance of any meta-element inadvertently. All of the keywords within the textual concrete syntax start with a lower-case letter. Therefore, a prevention mechanism is provided for preventing the users from defining a name starting with a lower-case for names which will not cause inconsistency between keywords and names. Validation Packages of Xtext are overridden for controlling user's variable definition. As illustrated in Listing 8, the editor will show an error if the developer defines a capability name starting with an upper-case. The corresponding code is written in the '*Validation Package*' in Xtext and some extra code is added to this package. These constraint controls are realized within the validation package (instead of grammar terminal rules) for enhancing the provision of customized error and warning messages, and also the possibility of fixing these errors and warnings. Similar controls are provided for other entities like Plan, SemanticWebAgent, SemanticWebService, etc.

```
01 @Check
02 public void CapabilitiesStartWithLowerCase(
03     Capabilities cap) {
04     if ( ! Character.isLowerCase(cap.getName().charAt(0)) ) {
05         error("Name must start with lower case",
06         AgentInternalDSLPackage. CAPABILITIES__NAME);
07     }
08 }
```

Listing 8. Validation Package code for preventing the definition of an upper-case name within the Semantic Web Agent Internal viewpoint

Additional Xtext features are used to limit the user whilst creating instance models, for example, another control supplied with the Validation Package code which prevents the user entering an empty string to an attribute. The code block in Listing 9 provides an error in the editor, if the user gives an empty string to the *'type'* attribute of a *Behavior*. Within the Xtext validation package, '*@Check*' is a java annotation for defining a validation rule.

```
01  @Check
02  public void checkTypeIsNotEmpty (Behavior beh)
03  {
04      if ( beh.getType().isEmpty() ) {
05           error("Behavior type is empty",
06           AgentInternalDSLPackage.BEHAVIOR__TYPE);
07      }
08  }
```

Listing 9. Validation Package code to prevent defining an empty string

```
01  public void checkNegativeElement (Plan plan)
02  {
03      If ( plan.getPriority ( ) < 0 )
04          error ( "Negative value is not accepted",
05              MyDslPackage.PLAN__DESCRIPTION);
06  }
```

Listing 10. Validation Package code to check the negative values for plan priority

In some part of the language, validity for a variable's value is examined using an overridden Xtext validation package. For example, as shown in Listing 10, the value of the priority for the plan element is checked, and negative values are not accepted.

3.4. Textual Concrete Syntax of Agent-Semantic Web Service Interaction Viewpoint

When considering Agent-SWS Interaction viewpoint, instances of related meta-elements and their relations must be defined inside a *SWSInteractionViewpoint* code-block as part of the instance model. Similar to the Agent Internal viewpoint, in this viewpoint, the left-hand bracket must be at the beginning of the model and the right-hand bracket at the end of it. Every instance model must have at least one SemanticWebAgent and one SemanticWebService, and every command or declaration must end with a semicolon. Otherwise, an error will occur in the editor. According to Figure 3, a SemanticWebService must have relations with Grounding, Process, and Interface. Each instance model must contain these elements and the relations between them. Part of the Xtext code for supplying these relations is given in Listing 11. Line 4 forces the user to use the *'describes'* relation. Lines 10, 11, and 16 have similar meanings.

Some rules are written in order to provide a specific sequence of code, while another group of rules allows them to be independent of a sequence within the textual instance model, where it is required. For example, Lines 10 and 11 are written to supply the independency within the sequence of relations in Listing 11. The user can define the '*supports SWS*' relation before or after a '*calls WebService*' relation. In addition, the user can define the '*calls WebService*' relation as often as necessary, whereas it is restricted to defining only one '*supports SWS*' relation.

According to the Agent-SWS Interaction viewpoint, each instance model should have at least one SemanticWebAgent and one SemanticWebService supplied with the '*Validation Package*'. Listing 12 shows the implementation of the *checkAtLeastOneSWS* constraint.

In Listing 12, Lines 4 to 8 capture the SemanticWebServices from the AgentSWSInteractionViewpoint and place them on a list (swslist). In Line 9, the size of the *'swslist'* is controlled. If there is no element within the list, the editor will show an error.

```
01  Process:
02      'Process' name=ID';'|
03      process=[Process]  '{'
04             'describes' sws=[SWS] ';'
05             …
06      '}';
07  Grounding:
08      'Grounding'  name = ID';' |
09      grounding  = [Grounding] '{'
10            ('supports' sws=[SWS] ';') &
11            ('calls' service += [WebService] ';')*
12      '}';
13  Interface:
14     'Interface'  name = ID ';' |
15      interface=[Interface]  '{'
16            'presents' sws=[SWS] ';'
17            …
18      '}';
```

Listing 11. Parts of Process, Grounding, and Interface rules

In regard to the constraints when creating plans, we can consider plan types in Agent-SWS Interaction viewpoint. According to SEA_L, textual concrete syntax, Semantic Service Plans (SS_RegisterPlan, SS_FinderPlan, SS_AgreementPlan and SS_ExecutorPlan), and their relations, must be in a specific order within the instance models. This order helps increasing readability of the program. These sequence restrictions are supplied with EBNF rules in Listing 13.

```
01  @Check
02  public void checkAtLeastOneSWS(
03              AgentSWSInteractionViewpoint sws) {
04       SWSInteractionViewpoint agent =
05              EcoreUtil2.getContainerOfType(sws,
06                   SWSInteractionViewpoint.class);
07       List<SWS> swslist =
08             EcoreUtil2.getAllContentsOfType(agent, SWS.class);
09       if((swslist.size()<1))
10             error("There must be at least one
11                  SWS", AgentSWSInteractionPackage.Literals.
12                     SWS_INTERACTION_VIEWPOINT__NAME);
13  }
```

Listing 12. Validation Package code for supplying at least one SWS constraint

According to Lines 2 and 3, any general Plan or Semantic Service Plan can be defined within the instance model. A Plan can be defined with or without its *'type'*, *'description'* and *'priority'* attributes. The ‘?’ character at the end of each statement makes it optional. If Semantic Service Plans are considered, the order should be as defined in Lines 5 to 8. In Line 11, it is

stated that one or more 'advertises interface' relation can exist. Similar rules are defined for other plan types in Lines 15-16, 20, and 24-25.

The Xtext can generate EBNF rules from a given metamodel. It can also generate a metamodel from the EBNF rules. However, we preferred to define EBNF rules manually in order to supply some syntactical restrictions and constraints such as defining relations in a specific order (Xtext cannot extract the order from the metamodel because the metamodel has not such an attribute by itself). It is worth noting that when starting from the already-existing metamodel and defining EBNF rules manually, one should be careful to properly match the metamodel with the grammar.

In this study, as mentioned previously, some controls are also used with a formatting package in addition to using some controls with a validation package. For example, some rules are defined for modifying the written program in order to rearrange the format of the code to gain more readability. Moreover, some other Xtext facilities are used, e.g. Wizard sample code, Highlighting (for keywords, comments, variables, etc), and Quick-fixing for errors.

```
01 Plan returns Plan:
02     ('Plan' name = ID (type=STRING)?
03     (description = STRING)?(priority=INT)? ';') | PlanSequence;
04 PlanSequence returns Plan:
05     reg  = SS_RegisterPlanDef
06     find = SS_FinderPlanDef
07     agree = SS_AgreementPlanDef
08     exe = SS_ExecutorPlanDef ;
09 SS_RegisterPlan:
10     plan=[SS_RegisterPlanDef]  '{'
11          ('advertises' interface+=[Interface] ';')+
12     '}';
13 SS_FinderPlan:
14     plan=[SS_FinderPlanDef] '{'
15          'interactsWith' matchmaker=[SSMatchmakerAgent]';'
16          ('discovers' interface+=[Interface]';')*
17     '}';
18 SS_AgreementPlan:
19     plan=[SS_AgreementPlanDef] '{'
20          'negotiates' interface=[Interface] ';'
21     '}';
22 SS_ExecutorPlan:
23     plan=[SS_ExecutorPlanDef]  '{'
24          'executes' process=[Process] ';'
25          'uses' grounding=[Grounding] ';'
26     '}';
```

Listing 13. Sample Plan rules

3.5. Code Generation

It is not sufficient to complete the DSL definition only by specifying the notions and their representations. A complete definition requires that one provides the semantics of language concepts in terms of other concepts, the meanings of which are already established. Therefore the syntax of the SEA_L is mapped into the metamodels of existing agent platforms that have well-defined, understood, and executable semantics. This mapping is provided through model transformations [5, 9, 31, 44]. Model to code transformations follow these model transformations and, finally, an executable software code is achieved for exact MAS.

In our study, code generation for the instance models is supplied with the Xpand tool [50]. Many of model driven engineering approaches accomplish code generation by writing strings to the text files. Xpand is a template engine which is used to make this process easier. It allows for creating textual output using EMF [10] models. The text output can be coded within any programming language. Xpand requires an EMF metamodel and one or more templates for translating the model into text. Once the requirements are provided, code generation can be provided by first defining an EMF model and running the generator. Xpand supplies traverse the abstract tree of the provided model and generate the code along the way [51].

In this study, Xpand is used for the generation of JADEX [23] code, along with OWL [36] and OWL-S [37] files from SEA_L specifications, and corresponding instance models. The code generation of JADEX agents from the SEA_L's Agent Internal viewpoint, and the generation of OWL-S SWS documents from SEA_L's Agent-SWS Interaction viewpoint, are exemplified in this paper.

JADEX is one of the popular APIs for developing software agents. JADEX code is composed of two files: the Agent Definition File (ADF), in which an agent's Beliefs, Goals, and Plans are defined using XML code, and the JADEX Plan File, in which Agent plans are defined using Java code. According to the JADEX platform, each agent has an ADF file. Therefore, in our study, an ADF file is generated for each SemanticWebAgent of a SEA_L instance model. The Beliefs, Goals, Plans, Behaviors, and Capabilities of SemanticWebAgents are defined within ADF with corresponding tags, but the JADEX Plan files include pure Java code for defining corresponding tasks.

In the generated code for SEA_L models, SEA_L ontological entities such as agent knowledge-bases are coded in OWL. Moreover, SWSs modeled in SEA_L instances are implemented according to OWL-S specifications. Both OWL and OWL-S are perhaps the most popular and in-use technologies for describing ontologies and SWS definitions.

An instance model, which conforms to the SEA_L metamodel, is in fact a platform independent model. In order to achieve its platform specific counterparts (e.g. its JADEX counterpart), mappings are needed between the SEA_L metamodel and metamodels of agent development frameworks (e.g., JADEX, JADE [22]). Since we focus on the JADEX platform in this study, we need to provide entity mappings between SEA_L and JADEX metamodels.

These mappings pave the way for transforming the source model (SEA_L) into the target model (JADEX). The mappings are illustrated in Table 1.

As discussed in subsection 3.1, the Agent Internal viewpoint focuses on the internal structure of every Agent within a MAS organization. Hence, in order to generate JADEX code, Agent Internal viewpoint is mapped to a JADEX metamodel. On the other hand, the Agent-SWS Interaction viewpoint represents the interaction between SemanticWebAgents and SemanticWebServices. Thus, it is mapped to both JADEX and OWL-S metamodels (see Table 1). The generated ontology files for Agent-SWS Interaction viewpoint are provided together with the ADF and Plan files for the Agent Internal viewpoint. Since the generations of ADF and Plan files for the Agent-SWS Interaction viewpoint are very similar to those for the Agent Internal viewpoint, it is not repeated here.

It is worth noting that both the mappings between SEA_L and JADEX and SEA_L and OWL-S take place simultaneously. In fact the SEA_L instance elements pertaining to agent and MAS viewpoints are transformed into JADEX instances while remaining elements of the same SEA_L instance model, which are used to model semantic web services, are transformed into OWL-S instances.

Table 1. Mapping between SEA_L, JADEX and OWL-S Metamodels

SEA_L	JADEX	OWL-S
SemanticWebAgent	Agent	
SSMatchmakerAgent	Agent	
Plan	Plan	
Behavior	Plan	
Capabilities	Capability	
Goal	AchieveGoal	
Goal	QueryGoal	
Goal	PerformGoal	
SS_AgreementPlan	Plan	
SS_ExecutorPlan	Plan	
SS_FinderPlan	Plan	
SS_RegisterPlan	Plan	
SemanticWebService		Service
Interface		ServiceProfile
Process		ServiceModel
Grounding		ServiceGrounding
Input		Input
Output		Output
Precondition		Condition
Effect		ResultVar

For code generation, a metamodel namespace is initially imported in order to make the meta-types known to the editor, as shown in Line 1 of Listing 14. Next, the main template is created. Each template is defined by a rule starting with the DEFINE keyword (see Line 2 of Listing 14). Xpand's

keywords and meta-type references are always enclosed in '«' and '»' characters.

```
01  «IMPORT org::xtext::example::mydsl::myDsl»
02  «DEFINE main FOR SWSInteractionViewpoint»
03  …
04  «EXPAND owlservice FOREACH service»
05  «EXPAND owlsprofile FOREACH service»
06  «EXPAND owlsmodel FOREACH service»
07  «EXPAND owlgrounding FOREACH service»
08  «EXPAND wsdl FOREACH service»
09  «ENDDEFINE»
```

Listing 14. Defining main elements and invoking templates

Each template consists of a template name and meta-type on which the template can be called. In this way a template is rather like a sub-routine, parameterized by a meta-type and other optional parameters [27]. So, in our study, model transformations are supplied in a built-in way between the SEA_L, JADEX, and OWL-S metamodels. For example, a SemanticWebAgent element in an instance model of SEA_L is transformed into a JADEX Agent element while generating the code. These transformations are supplied regarding the mappings in Table 1.

In Listing 14, for each Service, *'owlservice'*, *'owlsprofile'*, *'owlsmodel'*, *'owlsgrounding'*, and *'wsdl'* (Web Service Definition Language) templates are invoked between lines 4 to 8. Each SemanticWebService is represented in a '*Service.owl*' file. For example, for an '*Electronic Barter Service*', an '*EBarterService.owl*' file will be produced. '*Service Profile*', '*Service Process*' and '*Service Grounding*' are described within the '*profile.owl*', '*process.owl*' and '*grounding.owl*' files, respectively.

According to the second line of Listing 15, a '*Service.owl*' file is created. The other lines of the code are added to the end of this file. The bold keywords (*int*, *pro* and *gro*) are the predefined variables representing the Interface, Process, and Grounding, respectively. Lines 4, 7 and 11 are the point references for the Profile, ProcessModel and Grounding, respectively. Also, the related service name will be written in generated code by using '«this.name»' in Lines 3, 5, 9, and 13.

Nested templates are defined for invoking input, output, precondition, and effect where they are needed. In the Agent Internal viewpoint, an ADF file is needed for each SemanticWebAgent, and a Plan file is needed for each Plan. Therefore, the Plans and SemanticWebAgent templates are invoked within the main template, as represented in Listing 16.

Listing 17 shows the Xpand code for creating Plan files. Lines 3 to 22 are all boilerplate texts for inserting into the plan file.

The code-block given in Listing 18 represents the belief definitions, as a sample element, within the generated ADF file. Beliefs are defined in <beliefs> tags. The attributes of a belief meta-entity are generated using Lines 3-5 of Listing 18.

Code generations for other parts of ADF (e.g. Goal and Capability) are realized in a similar manner.

```
01  «DEFINE owlservice FOR Service»
02  «FILE this.name + "Service.owl"»
03  <service:Service rdf:ID= "«this.name»">
04      <!-- Reference to the Profile -->
05      <service:presents rdf:resource="&«this.name»_profile;#
06          «int.name»"/>
07      <!-- Reference to the Process Model -->
08      <service:describedBy
09          rdf:resource="&«this.name»_process;#
10          «pro.name»"/>
11      <!-- Reference to the Grounding -->
12      <service:supports
13          rdf:resource="&«this.name»_grounding;#
14          «gro.name»"/>
15  </service:Service>
```

Listing 15. A part of the Xpand code for defining the OWL-S Service File

```
01  «IMPORT  org::xtext::example::agentinternal::agentInternal»
02  «DEFINE   main  FOR AgentInternalViewpoint»
03  «EXPAND plans FOREACH plan»
04  «EXPAND semanticwebagents   FOREACH semanticwebagent»
05  «ENDDEFINE»
```

Listing 16. Sample template for invoking plans and semanticwebagents templates

Code generation for other viewpoints including the Environment, Role, Plan, and Interaction viewpoints are provided similarly. The required code generated from these viewpoints extend the agents' files, ADFs and plans, in the same way as Agent Internal and Agent-SWS Interaction viewpoints do.

As an expected result of applying MDD techniques, SEA_L simplifies the process of software development for MASs working within a semantic web environment. When considering the traditional approach for developing this type of software, a programmer should develop an ADF file (XML format) for each agent and a plan file (a Java file) for each plan of the agent, and then interconnect them. Also, the programmer should provide service, profile, grounding, process model, and WSDL documents for each semantic web service as required in the OWL-S standard. Meanwhile, the developer should consider the relation between these documents as well as the interaction between both the intra agents and agents with semantic web services. Therefore, the process is quite complex. However, in order to develop this type of software in SEA_L, the developer only needs to provide a program at the higher level (abstracting from the target platform constraints), which can help to produce all the above-mentioned documents and their interconnections, automatically.

```
01  «DEFINE plans FOR Plan»
02  «FILE name + ".java"»
03  import java.util.*;
04  import jadex.runtime.*;
05  import java.util.StringTokenizer;
06  public class «this.name»  extends Plan {
07      // Plan attributes.
08      ...
09      // static block or constructor
10      ...
11      // Constructor code.
12      public «this.name»() {
13                      ...
14      }
15      // Plan main code.
16      public void body() {
17              // Send request
18              ...
19              // Wait for reply
20               …
21      }
22  }
23  «ENDFILE»
24  «ENDDEFINE»
```

Listing 17. Xpand code to generate JADEX Plan files

```
01  «DEFINE beliefs FOR Belief»
02  <beliefs
03        Name = «this.name»
04        Description = «this.description»
05        dynamic = «this.dynamic»
06  />
07   «ENDDEFINE»
```

Listing 18. Sample Xpand code for defining beliefs in ADF

4. Related Work

Studies on DSLs and Domain-specific Modeling Languages (DSML) for agents have recently emerged, and these very few studies are still at their preliminary stages. For instance, a DSL called Agent-DSL is introduced in [28]. Agent-DSL is used to specify those agency properties that an agent should have in order to accomplish its tasks. However, the proposed DSL is only presented with its metamodel and just provides visual modeling of the agent systems according to agent features, such as knowledge, interaction, adaptation, autonomy, and collaboration. Likewise in [42], the authors introduced two dedicated modeling languages and call these languages

DSMLs. These languages are described by metamodels which can be seen as representations of the main concepts and the relations identified for each of the particular domains, again introduced in [42]. However, this study obviously included just the abstract syntax of the related DSMLs and does not give the concrete syntax or semantics of the DSMLs. In fact, the study only defines the generic agent metamodels for the MDD of MASs.

In [17], the author introduces a DSML for MAS. The abstract syntax of the DSML is derived from a platform independent metamodel, which is structured into several aspects each focusing on a specific viewpoint of a MAS. This approach is similar to our study. In order to provide the concrete syntax, the appropriate notations for the concepts and relations are defined in [49]. The semantics of the language is also given in [18]. These studies are noteworthy because they seem to provide the first complete DSML for agents, with all of its specifications. However it supports neither the agents on the Semantic Web nor the interaction of Semantic Web enabled agents with other environment members, such as semantic web services. Our study contributes to the aforementioned efforts by also specializing in the Semantic Web support of the MASs.

In [19], the authors introduce their approach on integrating agents with Semantic Web Services (SWSs) on a platform independent level. In addition to the MAS metamodel described in [17], a new platform independent metamodel is proposed for SWS. A relation between these two metamodels is established in a way that the MAS metamodel is extended with new meta-entities in order to support SWS interoperability and it also inherits some meta-entities from the metamodel proposed for SWS. Instead of using two separate metamodels, SEA_L has a built-in support for the modeling of agent and SWS interactions by including a special viewpoint. Moreover, semantic knowledge-base and agent internals can also be modeled in SEA_L.

Likewise, a new DSML is provided for MASs in [16]. The abstract syntax is presented using Meta-object Facility (MOF) [33] architecture. The concrete syntax and its tool are provided within a Graphical Modeling Framework (GMF) [11], and finally the code generation for the JACK agent platform [21] is realized by model transformations using JET [13]. However, the developed modeling language is not generic since it is only based on the metamodel of one of the specific MAS methodologies called Prometheus [38]. A similar study has been realized in [15] which proposes a technique for the definition of agent-oriented engineering process models and can be used for defining processes for creating both hardware and software agents. This study also offers the related MDD tool using Software & System Process Metamodel (SPEM) [34] and based on INGENIAS methodology [39] for MAS development. Nevertheless, similar to the DSML introduced in [17], neither [16] nor [15] cover software agents within the Semantic Web.

By considering our previous studies, in [25], we show how domain specific engineering can provide easy and rapid construction of Semantic Web enabled MASs. Ideas have been discussed for abstract syntax, concrete syntax, and formal semantics. Furthermore, a metamodel, which in fact constitutes the preliminary version of the abstract syntax of SEA_L, is

introduced in [4]. Based on these building blocks, in this paper we have discussed SEA_L by including its syntax and semantics definitions, and shown how the language and its tools can be used during the development of real MASs.

5. Conclusion

This paper discussed the textual concrete syntax of a new DSL, called SEA_L, for Semantic Web enabled MASs. Additionally, we showed how the specifications of SEA_L can be used during the development of real MASs. Hence, agent software developers can first design their MASs by only taking care of the MAS domain specifications and abstracting from the target platform constraints. Following this domain specific design, the automatic application of predefined transformations enables developers to achieve executable code for the agent system that is intended for implementation in target platforms such as JADEX. Apart from its unique support for the Semantic Web, the use of SEA_L also brings an easier way of MAS development compared to merely programming with JADEX or any other specific MAS development framework.

For the concrete syntax, meta-elements are mapped to textual notations in Xtext, textual constraints are provided, and verification of these constraints was shown within the instance models. In this way, we have provided an interpreter mechanism and created an automatic code generation for users of the domain using Xtext and Xpand tools. Transformations from SEA_L to the other MAS platforms, e.g. JADE and JACK, are aimed in the next step. Hence, our Xpand-based interpreter for SEA_L presented in this paper can also be used for the implementation of SEA_L instances in other MAS platforms in addition to the JADEX.

As future work, we aim to evaluate SEA_L by providing two groups of MAS programmers with the same programming ability and then give them a real problem which can be solved by agents working within a semantic web environment. The first group would apply the classical approach of agent programming within the JADEX platform and semantic web programming in OWL-S. The second group would use SEA_L language to develop the solution and later they would add a complementary code (in JADEX and OWL-S) to the generated code by SEA_L. Based on their results, we would compare the development time, the amount of errors occurring for both groups, and the quality of the final code, again for both groups. In addition, we would compare the ratio of generated code with the full final code for the performance evaluation of SEA_L.

Acknowledgment. This study is 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. Also, we gratefully acknowledge the helpful comments from anonymous referees.

References

1. Badica, C., Budimac, Z., Burkhard, H. D., and Ivanovic, M.: Software agents: Languages, tools, platforms. Computer Science and Information Systems, Vol. 8, No. 2, 255-298. (2011)
2. Bradshaw, J. M.: Software Agents. MIT Press Cambridge, MA, USA. (1997)
3. Bratman, M. E.: Intention, Plans, and Practical Reason. Harvard University Press, Cambridge, Massachusetts. (1987)
4. Challenger, M., Getir, S., Demirkol, S., and Kardas, G.: A Domain Specific Metamodel for Semantic Web enabled Multi-agent Systems. Lecture Notes in Business Information Processing, Vol. 83, 177-186. (2011)
5. Czarnecki, K., and Helsen, S.: Feature-Based Survey of Model Transformation Approaches. IBM Systems Journal - Model-driven software development, Vol. 45, Issue 3, 621-645. (2006)
6. Demirkol S., Challenger M., Getir S., Kosar, T., Kardas G. and Mernik, M.: SEA_L: A Domain-specific Language for Semantic Web enabled Multi-agent Systems. Second Workshop on Model Driven Approaches in System Development (MDASD 2012), held at Federated Conference on Computer Science and Information Systems (FedCSIS 2012), Wrocław-Poland, 9-12 September, 1373-1380. (2012)
7. Demirkol, S., Getir, S., Challenger M., and Kardas, G.: Development of an Agent based E-barter System. In International Symposium on Innovations in Intelligent Systems and Applications (INISTA), IEEE Computer Society, 193-198. (2011)
8. van Deursen, A., Klint, P., and Visser, J.: Domain-specific Languages: an annotated bibliography. ACM SIGPLAN Notices, Vol. 35, No. 6, 26-36. (2000)
9. Duddy, K., Gerber A., Lawley, M., Raymond, K. and Steel, J.: Model Transformation: A Declarative, Reusable Patterns Approach. In Azada, D. (Ed.) proceedings of Seventh IEEE International Enterprise Distributed Object Computing Conference (EDOC'03), IEEE Computer Society, Brisbane, Australia, 174-185. (2003)
10. Eclipse EMF: [Online] Available: http://www.eclipse.org/modeling/emf (Last access: March 2013)
11. Eclipse GMF: [Online] Available: http:// www.eclipse.org/modeling/gmp/ (Last access: March 2013)
12. Eclipse Help for Xtext: [Online] Available: http://help.eclipse.org/helios/index.jsp (Last access: March 2013)
13. Eclipse JET: [Online] Available: http://www.eclipse.org/modeling/m2t/?project=jet (Last access: March 2013)
14. Fowler, M.: Domain-specific Languages. Addison Wesley. (2011)
15. Fuentes-Fernandez, R., Garcia-Magarino, I., Gomez-Rodriguez, A. M., and Gonzalez-Moreno, J. C.: A Technique for Defining Agent-Oriented Engineering Processes with Tool Support. Engineering Applications of Artificial Intelligence, Vol. 23, Issue 3, 432–444. (2010)
16. Gascuena J. M., Navarro, E., and Caballero, A. F.: Model-Driven Engineering Techniques for the Development of Multi-agent Systems. Engineering Applications of Artificial Intelligence, Vol. 25, 159–173. (2012)
17. Hahn, C.: A Domain Specific Modeling Language for Multi-agent Systems. In Seventh International Conference on Autonomous Agents and Multi-agent Systems (AAMAS'08), ACM Press, 233-240. (2008)

18. Hahn C., and Fischer K.: The Formal Semantics of the Domain Specific Modeling Language for Multi-agent Systems. Lecture Notes in Computer Science, Vol. 5386, 145-158. (2009)
19. Hahn, C., Nesbigall, S., Warwas, S., Zinnikus, I., Fischer, K., and Klusch, M.: Integration of Multi-agent Systems and Semantic Web Services on a Platform Independent Level. In IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology, 200-206. (2008)
20. ISO/IEC 14977:1996 Standard, Information technology, Syntactic meta language - Extended BNF.
21. JACK: [Online] Available: http://aosgrp.com/products/jack/ (Last access: March 2013)
22. JADE: Java Agent DEvelopment Framework. [Online] Available: http://jade.tilab.com/ (Last access: March 2013)
23. JADEX: [Online] Available: http://jadex-agents.informatik.uni-hamburg.de/xwiki/bin/view/About/Overview (Last access: March 2013)
24. Kardas G., Challenger M., Yildirim S., and Yamuc A.: Design and Implementation of a Multi-agent Stock Trading System. Software: Practice and Experience, Vol. 42, Issue 10, 1247–1273. (2012)
25. Kardas, G., Demirezen, Z., and Challenger, M.: Towards a DSML for Semantic Web enabled Multi-agent Systems. In International Workshop on Formalization of Modeling Languages, held in conjunction with the Twenty fourth European Conference on Object-Oriented Programming (ECOOP2010), ACM Press, 1-5. (2010)
26. Kos, T., Kosar, T., Knez J., and Mernik, M.: From DCOM interfaces to domain-specific modeling language: A case study on the Sequencer. Computer Science and Information Systems, Vol. 8, No. 2, 361-378. (2011)
27. Koster, V.: Implementation and Integration of a Domain Specific Language with oAW and Xtext. Technical Report. (2007)
28. Kulesza, U., Garcia, A., Lucena C., and Alencar, P.: A Generative Approach for Multi-agent System Development. Lecture Notes in Computer Science, Vol. 3390, 52-69. (2005)
29. Liu, S-H., Cardenas, A., Mernik, M., Bryant, B. R., Gray, J., and Xiong, X.: Introducing Domain-specific Language Implementation Using Web-Service Oriented Technologies. Multiagent and Grid Systems, Vol. 8, 19-44. (2012)
30. Macikenas, E., and Makunaite, R.: Applying Agent in Business Evaluation Systems. Information Technology and Control, Vol. 37, No. 2, 101 – 105. (2008)
31. Mens, T., and Van Grop, P.: A Taxonomy of Model Transformation. Electronic Notes in Theoretical Computer Science, Vol. 152, 125-142. (2005)
32. Mernik, M., Heering, J., and Sloane, A.: When and how to develop domain-specific languages. ACM Computing Surveys, Vol. 37, No. 4, 316-344. (2005)
33. Object Management Group, Meta Object Facility (MOF) 2.0 Core Specification. [Online] Available: www.omg.org/spec/MOF/2.0/ (Last access: March 2013)
34. Object Management Group, Software & System Process Engineering Metamodel Specification Version 2.0, formal/2008-04-01, 2008. [Online] available at: http://www.omg.org/spec/SPEM/2.0/ (Last access: March 2013)
35. OMG ODM: [Online] Available: http://www.omg.org/spec/ODM/1.0/ (Last access: March 2013)
36. OWL: [Online] Available: http://www.w3.org/TR/owl-features (Last access: March 2013)
37. OWL-S: [Online] Available: http://www.w3.org/Submission/OWL-S/ (Last access: March 2013)

38. Padgham, L., and Winikoff, M.: Developing Intelligent Agent Systems: A Practical Guide. John Wiley & Sons, Ltd Publications. (2004)
39. Pavon, J., Gomez-Sanz, J. J., and Fuentes, R.: The INGENIAS Methodology and Tools. In Henderson-Sellers, B., Giorgini, P. (Eds.), Agent-Oriented Methodologies, Article IX. Idea Group Publishing, 236–276. (2005)
40. Pešović D., Vidaković M., Ivanović M., Budimac Z. and Vidaković J.: Usage of Agents in Document Management. Computer Science and Information Systems, Vol. 8, No. 1, 193-210. (2011)
41. Rao, A., and Georgeff, M.: BDI Agents: From Theory to Practice. In Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95), 312-319, San Francisco. (1995)
42. Rougemaille, S., Migeon, F., Maurel, C., and Gleizes, M-P.: Model Driven Engineering for Designing Adaptive Multi-agent Systems. Lecture Notes in Artificial Intelligence, Vol. 4995, 318-33. (2007)
43. Schmidt, D. C.: Guest Editor's Introduction: Model-Driven Engineering. IEEE Computer, Vol. 39, No. 2, 25-31. (2006)
44. Sendall, S., and Kozaczynski, W.: Model Transformation: the Heart and Soul of Model-Driven Software Development. IEEE Software, Vol. 20, Issue 5, 42-45. (2003)
45. Shadbolt, N., Hall, W., and Berners-Lee, T.: The Semantic Web Revisited. IEEE IEEE Intelligent, Vol. 21, Issue. 3, 96-101. (2006)
46. Sycara, K.: Multi-agent Systems. AI Magazine, Vol. 19, 79-92. (1998)
47. Vallecillo, A.: A Journey through the Secret Life of Models. In Perspectives Workshop: Model Engineering of Complex Systems (MECS), 08331 in Dagstuhl Seminar Proceedings, Germany. (2008)
48. Varanda-Pereira, M. J., Mernik, M., Da Cruz, D., and Henriques, P. R.: Program Comprehension for Domain-specific Languages. Computer Science and Information Systems, Vol. 5, No. 2, 1-17, (2008)
49. Warwas S., and Hahn, C.: The Concrete Syntax of the Platform Independent Modeling Language for Multi-agent Systems. In Agent-based technologies and applications for enterprise interoperability, held in conjunction with the Seventh International Conference on Autonomous Agents and MASs, AAMAS. (2008)
50. Xpand: [Online] Available: http://wiki.eclipse.org/Xpand (Last access: March 2013)
51. Xpand documentation: [Online] Available: http://ditec.um.es/ssdd/xpand_reference.pdf (Last access: March 2013)
52. Xtext: [Online] Available: http://www.eclipse.org/Xtext/ (Last access: March 2013)

Sebla Demirkol received her B.Sc in Mathematics (Computer Science division) and M.Sc in Information Technologies from Ege University in 2009 and 2012 respectively. She is currently working as an Assistant Project Manager in Veripark Software Company. Her main research interests are model-driven development, multi agent systems and domain-specific languages.

Moharram Challenger received his B.Sc., and M.Sc. degrees in computer engineering from IAU-Shabestar and IAU-Arak Universities (Iran) in 2001 and 2005 respectively. Since 2006, he has been a tenure-track faculty member, as a lecturer, at computer engineering department, IAU-Shabestar University.

He is currently a Ph.D. candidate at Ege University, International Computer Institute with expected graduation of 2013. His research interests include domain-specific (modeling) languages, multi-agent and distributed systems with a current focus on the semantics of DSMLs. Moharram is also a student member of the IEEE and ACM.

Sinem Getir received her B.Sc in Mathematics (Computer Science division) and M.Sc in Information Technologies from Ege University in 2009 and 2012 respectively. She is currently a research assistant and a Ph.D. candidate in the University of Stuttgart. Her main research interests are model-driven development, formal semantics, model checking, and run-time verification. Other research interests are multi-agent systems and self-adaptive systems.

Tomaž Kosar received the Ph.D. degree in computer science at the University of Maribor, Slovenia in 2007. His research is mainly concerned with design and implementation of domain-specific languages. Other research interest in computer science include also domain-specific modelling languages, empirical software engineering, software security, generative programming, compiler construction, object oriented programming, object-oriented design, refactoring, and unit testing. He is currently a teaching assistant at the University of Maribor, Faculty of Electrical Engineering and Computer Science.

Geylani Kardas received his B.Sc. in computer engineering and both M.Sc., and Ph.D. degrees in information technologies from Ege University in 2001, 2003 and 2008 respectively. He is currently an assistant professor at Ege University, International Computer Institute. His research interests include model-driven software development, domain-specific (modeling) languages, agent-oriented software engineering and the Semantic Web. He is a member of the ACM.

Marjan Mernik received his M.Sc., and Ph.D. degrees in computer science from the University of Maribor in 1994 and 1998 respectively. He is currently a professor at the University of Maribor, Faculty of Electrical Engineering and Computer Science. He is also a visiting professor at the University of Alabama in Birmingham, Department of Computer and Information Sciences, and at the University of Novi Sad, Faculty of Technical Sciences. His research interests include programming languages, compilers, domain-specific (modeling) languages, grammar-based systems, grammatical inference, and evolutionary computations. He is a member of the IEEE, ACM and EAPLS.

Received: November 5, 2012; Accepted: April 29, 2013