SEA_L: A Domain-specific Language for Semantic Web enabled Multi-agent Systems

Sebla Demirkol[1] sebla.demirkol@ege.edu.tr
Moharram Challenger[1] moharram.challenger@mail.ege.edu.tr
Sinem Getir[1] sinem.getir@ege.edu.tr
Tomaž Kosar[2] tomaz.kosar@uni-mb.si
Geylani Kardas[1] geylani.kardas@ege.edu.tr
Marjan Mernik[2] marjan.mernik@uni-mb.si

[1]International Computer Institute, EGE University, Bornova, 35100 Izmir, Turkey
[2]Faculty of Electrical Engineering and Computer Science, University of Maribor, Maribor, Slovenia

Abstract— **Autonomous, reactive and proactive features of software agents make development of agent-based software systems complex. A Domain-specific Language (DSL) can provide the required abstraction and hence support a more fruitful methodology for the development of Multi-agent Systems (MASs) especially working on the new challenging environments such as the Semantic Web. Based on our previously introduced domain-specific metamodel, in this paper we propose a textual concrete syntax of a DSL for MASs working on the Semantic Web and show how the specifications of this DSL can be utilized during the code generation of exact MASs. The new DSL is called Semantic web Enabled Agent Language (SEA_L). The syntax of SEA_L is supported with textual modeling toolkits developed with Xtext. The practical use of SEA_L is illustrated with a case study which considers the modeling of a multi-agent based e-barter system.**

Keywords— Domain-specific Languages; Metamodel; Multi-agent Systems; Semantic Web

I. INTRODUCTION

SOFTWARE agents [1] are autonomous software components which are able to act on behalf of their users in order to do a group of defined tasks. Many intelligent software agents interact with each other in a system that is called Multi-agent System (MAS). Their interactions can be either cooperative or selfish [2]. 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, Semantic Web improves World Wide Web such that web page contents can be interpreted with ontologies [3]. 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) to handle the semantic content on behalf of their users. Surely, Semantic Web environment has specific architectural entities and a different semantic which must be considered to model a MAS within this environment. Thus, Semantic Web evolution brought a new vision into agent research. Software agents are planned to collect Web content from diverse sources, process the information and exchange the results. Autonomous agents can also evaluate semantic data and collaborate with semantically defined entities of the Semantic Web such as semantic web services by using content languages. However, considering agent interactions with Semantic Web elements adds more complexity for designing and implementing those systems.

On the other hand, Model Driven Development (MDD) is one of the important software development approaches, moving software development from code to models [4] which increases productivity [5] and reduces development costs [6]. Design and implementation of a MAS may become more complex when new requirements and interactions for new agent environments such as Semantic Web are considered. MDD can provide an infrastructure that simplifies the development of such MASs. To work in a higher abstraction level is of critical importance for the development of MASs since it is almost impossible to observe code level details of the MASs due to their internal complexity, distributedness and openness. Hence, such MDD application can increase the abstraction level in MAS development. MDD uses different approaches to realize its goals. One of these methods is Domain Specific Language (DSL) development [7, 8, 9, 10, 11]. DSLs are languages which comprise a domain's concepts and terminologies to supply the requirements of the domain. A DSL allows end-user programmers (domain experts) to describe the essence of a problem with abstractions related to a domain specific problem space.

A domain specific metamodel for semantic web enabled MASs is discussed in [12]. Based on this metamodel, in this paper, we present the textual concrete syntax of a DSL and discuss transformations required for code generation from the specifications of this DSL. We call this new DSL as Semantic web Enabled Agent Language (SEA_L).

The rest of this paper is organized as follows: Related work is given in Section 2. The abstract syntax and the textual concrete syntax of SEA_L are discussed in Sections 3 and 4 respectively. In section 5, the code generation mechanism for new DSL is illustrated. Section 6 includes a case study on the development of a MAS by using SEA_L. Finally, Section 7 concludes the paper and states the future work.

 This study is funded as a bilateral project by The Scientific and Technological Research Council of Turkey (TUBITAK) under grant 109E125 and Slovenian Research Agency (ARRS) under grant BI-TR/10-12-004.

II. RELATED WORK

The studies on DSLs and Domain-specific Modeling Languages (DSML) for agents are recently emerging and those very few studies are in their preliminary states. For instance, a DSL called Agent-DSL is introduced in [13]. Agent-DSL is used to specify the agency properties that an agent could have in order to accomplish its tasks. However, the proposed DSL is presented only with its metamodel and provides just the visual modeling of the agent systems according to agent features, such as knowledge, interaction, adaptation, autonomy and collaboration. Likewise in [14], the authors introduce two dedicated modeling languages and call those languages as DSMLs. The 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 [14]. However, the study obviously includes 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 generic agent metamodels for MDD of MASs.

In [15], 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. That approach resembles to our study. This study is noteworthy because it seems to be 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 aforementioned efforts by also specializing on the Semantic Web support of the MASs. In [16], 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 [15], a new platform independent metamodel for SWS is proposed. 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 the built-in support for the modeling of agent and SWS interactions by including a special viewpoint. Moreover, semantic knowledgebase and agent internals can also be modeled in SEA_L.

III. 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 how the models should look like.

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 SemanticWebService using content language.

The Platform Independent Metamodel (PIMM) which represents the abstract syntax of SEA_L is divided into eight viewpoints to provide clear understanding and efficient use. These viewpoints are: Agent Internal viewpoint, MAS viewpoint, Plan viewpoint, Role viewpoint, Interaction viewpoint, Environment viewpoint, Ontology viewpoint and Agent-Semantic Web Service (SWS) Interaction viewpoint. Discussion on whole metamodel can be found in [12]. We only concentrate on Agent Internal viewpoint as well as Agent-SWS Interaction viewpoint throughout this paper due to space limitations and the importance of these viewpoints.

Agent Internal viewpoint is related to the internal structure of semantic web agents and defines entities and their relations required for the construction of agents.

SemanticWebAgent (SWA) in the SEA_L abstract syntax stands for each agent in Semantic Web enabled MAS. A SemanticWebAgent is an autonomous entity which is capable of interaction with both other agents and SemanticWebServices within the environment.

SEA_L's metamodel (hence abstract syntax) supports both reactive and Belief-Desire-Intention (BDI) agent architectures. BDI was first proposed by Bratman [17] and used in 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 deliberative attitudes of agents, include the agent planning mechanism in order to achieve goals. Taking concrete BDI agent frameworks (such as JADEX [18] and JACK [19]) into consideration, we propose an entity called Capabilities which includes each agent's Goals, Plans and Beliefs about the surrounding.

Agent-SWS Interaction viewpoint focuses on the internal structure of SemanticWebServices and interaction of any SemanticWebAgent with SemanticWebServices in a MAS organization. Concepts and their relations for appropriate service discovery, agreement with the selected service and execution of the service are all defined in this viewpoint. Partial metamodel which represents this viewpoint is shown in Fig. 1. In this figure, elements filled with light gray come from other viewpoints which are shown on top or bottom of the element using "<<" and ">>". In other words, these elements are common elements among viewpoints and they tailor the viewpoints to each other.

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 in this metamodel. Agents use SS_RegisterPlan for communication with a service register to discover service capabilities. Other Plans are used to discover SemanticWebServices dynamically, call the services, get agreement with them and execute them, respectively.

SWS modeling approaches (i.e. OWL-S [20]) 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. "Process Model" defines service's internal combinations and service executor dynamics. Finally, "Physical Grounding" defines the service's executor protocol. These meta-entities are shown in Fig. 1 with Interface, Process and Grounding entities respectively. These components can use Input, Output, Precondition and Effect which are extensions of Web Ontology Language (OWL) Class from Object Management Group's (OMG) Ontology Definition Metamodel (ODM) [21].

Considering the other viewpoints of the SEA_L, 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 in this viewpoint. Role viewpoint shows distinct types of role. Agents can use several roles at any time and can alter these roles over the time. Interaction viewpoint focuses on agent communications and interactions in a MAS and defines entities and relations such as Interaction, Message, and MessageSequence. Environment viewpoint focuses on the relations between agents and to what they access. Environment contains all non-Agent Resources, Facts and Services. Ontology viewpoint brings all ontology sets and ontological concepts together. ODM OWL [22] Ontology from OMG is a standard for all of our ontology sets such as Role, Organization and ServiceOntologies.

IV. TEXTUAL CONCRETE SYNTAX

The textual concrete syntax of SEA_L is provided with Xtext [23]. In this paper, we focus only on Agent Internal and Agent-SWS Interaction viewpoints. Xtext is a language development framework to provide textual modeling languages. It can be used for creating a sophisticated Eclipse-based development environment. Xtext is based on EBNF (Extended Backus–Naur Form) [24] rules.

If the metamodel which represents the abstract syntax for SEA_L is considered as analysis phase of the concrete syntax of SEA_L, the design phase will be the part in which EBNF rules are described. One of the main advantages of DSLs is to validate domain specific constraints. The constraints of the language can be implemented with "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. 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 to properly indent the documents. By defining EBNF rules, above discussed constraints of SEA_L's metamodel are realized. With these capabilities, the new DSL possesses both structure and static semantics of the MAS domain. The structure is defined by the method signatures and the semantics are defined by constraint code.

A. *Textual Concrete Syntax of Semantic Web Agent Internal Viewpoint*

An Xtext grammar is structured with rules which are identified by text to the left of a colon. There is at least one

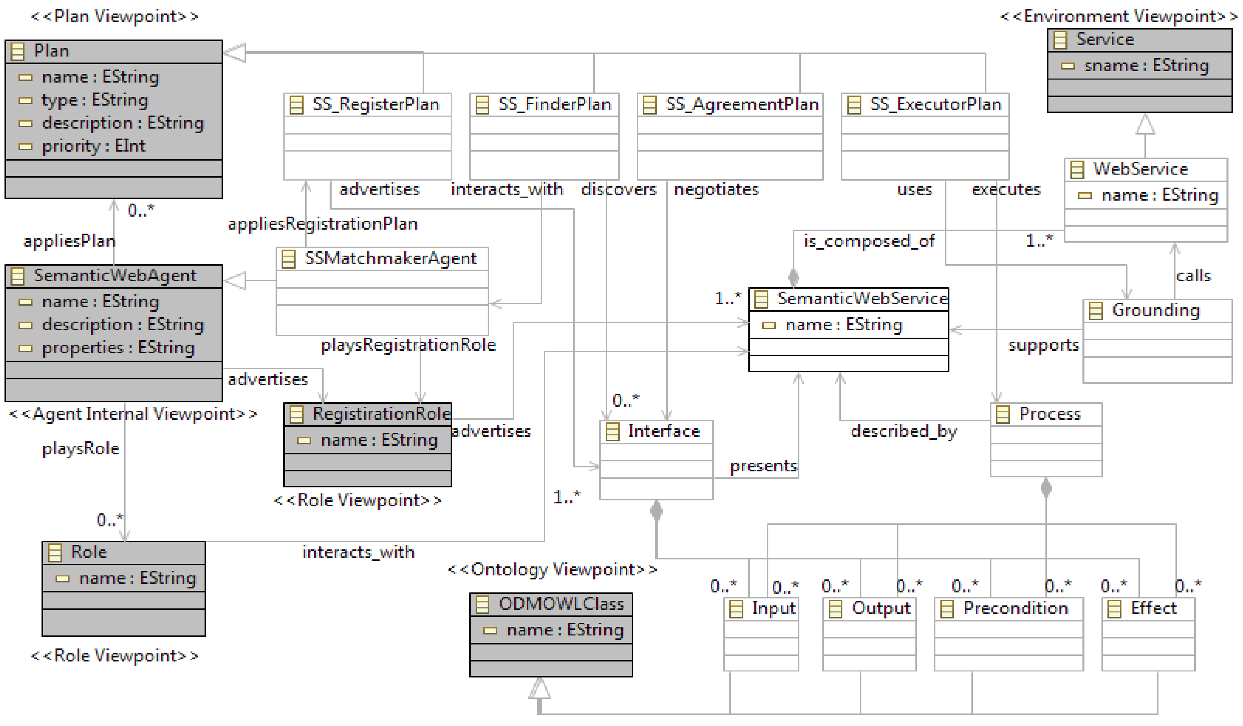

Fig. 1 Agent-SWS Interaction viewpoint.

rule for each meta-element in the textual concrete syntax. EBNF rules are defined for Agent Internal viewpoint according to constraints in the metamodel. The first constraint is that all of the elements of the instance model must be in “AgentInternalViewpoint” tag. Also, instance model must start and end with curly brackets. Example to another constraint is that each instance model must have at least one SemanticWebAgent and one Capabilities meta-entity in any order.

Each agent has at most one AgentType in the instance model. If a user defines more than one agent type for the SemanticWebAgent, the tool will give an error which is provided in SemanticWebAgent rule.

According to Xtext syntax, the assignment operator, "=", denotes a single valued feature, "+=" operator denotes a multi valued feature and the asterisk operator, "*", denotes a cardinality of 0..n. Also, in each rule, referring to predefined variables is possible with ‘[’ and ‘]’ characters as it is shown in Listing 1 line3.

SEA_L’s metamodel is based on BDI [17] architecture. Therefore, a group of meta-elements exist to supply BDI structure. Considering this structure, a Capabilities meta-element consists of Belief, Goal and Plan meta-elements. User can define numerous relations by considering the Agent Internal viewpoint. This structure is defined in Capabilities rule which is shown in Listing 1. Developer can define Belief, Goal and Plan meta-elements as much as needed in any order regarding lines 4 to 7 of Listing 1.

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

Listing 1. Capabilities’ rule.

Fewer constraints are defined in Agent Internal viewpoint in comparison to Agent-SWS Interaction viewpoint since elements are generally used arbitrary and most of the relations are independent in Agent Internal viewpoint.

Additional Xtext features are used to limit the user while creating instance models. Due to space limitation, a small example is given in Listing 2 which is written in the “Validation Package” of Xtext. Listing 2 provides an error in the editor, if the user gives an empty string to “type” attribute of a Behavior.

```
@Check
public void checkTypeIsNotEmpty (Behavior beh) {
   if ( beh.getType().isEmpty() ) {
       error("behavior type empty",
       AgentInternalDSLPackage.BEHAVIOR__TYPE);
   }
}
```

Listing 2. Validation Package code to prevent defining an empty string.

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

Considering Agent-SWS Interaction viewpoint, all of its instance models must be written inside a “SWSInteractionViewpoint” tag and every command or declaration must end with a semicolon. Otherwise, an error will occur in the editor. According to Fig. 1, a SemanticWebService must have relationships with Grounding, Process and Interface. Each instance model must contain these elements and relations between them. Part of Xtext code to supply these relations is given in Listing 3. Line 4 forces the user to use “described_by” relation. Lines 10 to 13 and 18 have similar meanings.

```
01 Process:
02    'Process' name=ID';'|
03    process=[Process]  '{'
04          'described_by' sws=[SWS] ';'
05          …
06    '}';
07 Grounding:
08    'Grounding'  name = ID';' |
09    grounding  = [Grounding] '{'
10        (('supports' sws=[SWS] ';')
11        ('calls' service = [WebService] ';') )|
12        (('calls' service = [WebService] ';')
13        ('supports' sws=[SWS] ';'))
14    '}';
15 Interface:
16   'Interface'  name = ID ';' |
17    interface=[Interface]  '{'
18        'presents' sws=[SWS] ';'
19        …
20    '}';
```

Listing 3. Parts of Process, Grounding, and Interface rules.

According to Agent-SWS Interaction viewpoint, each instance model should have at least one SemanticWebAgent and one SemanticWebService which is supplied with “Validation Package”. Listing 4 shows the implementation of the “checkAtLeastOneSWS” constraint.

```
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 4. Validation Package code to supply at least one SemanticWebService constraint.

In Listing 4, “@Check” is a keyword to define a validation rule. Lines 4 to 8 capture SemanticWebServices from the AgentSWSInteractionViewpoint and put them in a list (swslist). In line 9, size of "swslist" is controlled. If there is not any element in the list, the editor will give an error.

Some rules are written in order to provide a specific sequence for code while another group of rules let them to be independent of a sequence in textual instance model where it is required. For example, lines from 10 to 13 are written to supply the sequence independency of relations in Listing 3.

Semantic Service Plans (SS_RegisterPlan, SS_FinderPlan, SS_AgreementPlan and SS_ExecutorPlan) and their relations must be in a specific order in the instance models. For instance, SS_RegisterPlan must advertise an Interface before SS_FinderPlan interacts with SSMatchmaker Agent. These sequence restrictions are supplied with EBNF rules. Listing 5 shows how to supply Plan orders. According to lines 2 to 3, any general Plan or Semantic Service Plan can be defined in 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 as optional. If Semantic Service Plans are considered, the order should be as is defined in lines 5 to 8.

Xtext can generate EBNF rules from a given metamodel but we prefer to define EBNF rules manually to supply some preferred syntactical restrictions and constraints such as defining relations in a specific order (Xtext cannot extract the order from the metamodel because metamodel has not such an attribute by itself), defining at least one or more than one relation, etc.

```
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 ;
```

Listing 5. Sample Plan rules.

V. CODE GENERATION

It is not sufficient to complete the DSL definition only by specifying the notions and their representations. The complete definition requires that one provide semantics of language concepts in terms of other concepts whose meaning is 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. The mapping is provided through model transformations. Model to code transformations follow these model transformations and finally executable software code for exact MAS are achieved.

Code generation for the instance models are supplied with Xpand tool [25]. Many of model driven engineering approaches accomplish code generation by writing strings to text files. Xpand is a template engine which is used to make this process easier. It allows creating textual output using EMF [26] models. The text output can be coded in any programming language. Xpand requires an EMF metamodel and one or more templates to translate the model into text. Once the requirements are provided, code generator can be run by defining an EMF model and running the generator [27]. Xpand supplies traversing the abstract tree of provided model and generating the code along the way [27].

In this study, Xpand is used for generation of JADEX [18] code, along with OWL [21] and OWL-S [20] files from SEA_L specifications and corresponding instance models. Again due to space limitations, only code generation of JADEX agents from SEA_L Agent Internal viewpoint and generation of OWL-S SWS documents from Agent-SWS Interaction viewpoint are illustrated in this paper.

JADEX is one of the popular Application Programming Interfaces (API) for developing software agents. JADEX code is composed of two files: ADF (Agent Definition File), which an agent's Beliefs, Goals, and Plans are defined with XML code, The JADEX Plan File, which Agent plans are defined with Java code. According to JADEX platform, each agent has an ADF file. Therefore, an ADF file is generated for each SemanticWebAgent of a SEA_L instance model in our study. Beliefs, Goals, Plans, Behaviors and Capabilities of SemanticWebAgents are defined in ADF with corresponding tags, but JADEX Plan files include pure Java code 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 SEA_L metamodel, is in fact a platform independent model. In order to achieve its platform specific counterparts (e.g. its JADEX counterpart), mappings between SEA_L metamodel and metamodels of agent development frameworks (such as JADEX, JADE [28], etc.) are needed. Since we focus on JADEX platform in this study, we need to provide entity mappings between SEA_L and JADEX metamodels. These mappings are given in Table I.

As discussed in Section 3, Agent Internal viewpoint focuses on the internal structure of every Agent in a MAS organization. Hence, in order to generate JADEX code, Agent Internal viewpoint is mapped to JADEX metamodel. On the other hand, Agent-SWS Interaction viewpoint represents the interaction between SemanticWebAgents and SemanticWebServices. Thus, it is mapped to both JADEX and OWL-S metamodels (see Table I). Generated ontology files for Agent-SWS Interaction viewpoint and ADF and Plan files for Agent Internal viewpoint are provided. Since generation of ADF and Plan files for Agent-SWS Interaction viewpoint is very similar to the ones for Agent Internal viewpoint, it is not repeated here. It is also worth noting that both mappings between SEA_L and JADEX and SEA_L and OWL-S take place simultaneously such that 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.

Initially, metamodel namespace is imported in order to make the meta-types known to the editor as it is shown in line 1 of Listing 6. Next, the main template is created.

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

TABLE I. MAPPING BETWEEN SEA_L, JADEX AND OWL-S METAMODELS.

SEA_L	***JADEX***	***OWL-S***
SemanticWebAgent	Agent	
SSMatchmakerAgent	Agent	
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

```
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 6. Defining main elements and invoking templates.

In Listing 6, 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 in "profile.owl", "process.owl" and "grounding.owl" files respectively.

According to second line of Listing 7, a "Service.owl" file is created. The other lines of the code are added to the end of this file. Bold keywords ("int", "pro" and "gro") are predefined variables representing Interface, Process and Grounding respectively. Lines 4, 7 and 10 are point references to the Profile, ProcessModel and Grounding, respectively.

Nested templates are defined to invoke input, output, precondition and effect where they are needed. In Agent Internal viewpoint, an ADF file is needed for each SemanticWebAgent and a Plan file is needed for each Plan. Therefore, Plans and SemanticWebAgent templates are invoked in main template as it is represented in Listing 8.

Code block given in Listing 9 represents belief definitions in the generated ADF file. Beliefs are defined in <beliefs> tags. Attributes of a belief meta-entity are generated using line 3, 4 and 5 of Listing 9.

```
01  «DEFINE owlservice FOR Service»
02  «FILE this.name + "Service.owl"»
03  <service:Service rdf:ID= "«this.name»">
04     <service:presents rdf:resource="&«this.name_profile;#
05         "«int.name»"/>
06     <service:describedBy
07          rdf:resource="&"«this.name»_process;#
08          "«pro.name»"/>
09     <service:supports
10         rdf:resource="&"«this.name»_grounding;#
11         "«gro.name»"/>
12  </service:Service>
```

Listing 7. A part of Xpand code to define OWL-S Service File.

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

Listing 8. Sample template to invoke plans and semanticwebagents templates.

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

Listing 9. Sample Xpand code to define beliefs in ADF.

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

VI. CASE STUDY: E-BARTER SYSTEM

In order to illustrate the use of the introduced DSL, we consider the modeling of a simple multi-agent based e-barter system. A barter system is an alternative commerce approach where customers meet at a marketplace in order to exchange their goods or services without currency.

An agent-based e-barter system consists of agents that exchange goods or services of owners corresponding to their preferences. In this application, base scenario is achieved by the Customer, "Barter Manager" and Cargo roles assigned to the agents. Interested readers may refer to [29] for the detailed discussion of barter proposals and tracking the bargaining process between Customer agents. After finalization of bargaining, Customer agents send engagement message to the “Barter Manager” agent. Then, the “Barter Manager” agent notifies the Cargo agent for transporting barter products between Customer agents. This scenario is completed by the acceptance of all participating agents.

The following examples can be instances for the constraint controls in this case study:

Listing 10 illustrates the use of the Xtext editor in textual modeling of Agent-SWS Interaction viewpoint of the multi-agent e-barter system in question. In order to infer about semantic closeness between offered and purchased items based on the defined ontologies, barter manager may use a SemanticWebService called “Barter Service”.

As it is restricted in textual concrete syntax, each instance model must have at least one SemanticWebAgent and one SemanticWebServices. After declarations, “Barter Manager”, which is a SemanticWebAgent, applies SS_FinderPlan and SS_ExecutorPlan and plays Roles. SS_FinderPlan interacts with SS_MatchmakerAgent and gets the results of appropriate services. After this interaction, according to the results, SS_FinderPlan discovers “Barter ServiceInterface”.

```
SWSInteractionViewPoint  case2  {
    SemanticWebAgent barterManager "This is bartermanager
            agent" "Properties" ;
    SWS barterService ;
    SSMatchmakerAgent barterMatchAgent "Description"
            "Properties";
    Grounding barterServiceGrounding;
    Process barterServiceProcess;
    Interface barterServiceInterface;
    SS_RegisterPlan ServiceRegistertion;
    SS_FinderPlan discoverBarterService;
    SS_AgreementPlan Negotiating;
    SS_ExecutorPlan invokeBarterService;
    Role Registery;
    barterManager {
            appliesPlan discoverBarterService;
            appliesPlan ServiceRegistertion ;
            playsRole Registery;}
    barterMatchAgent {
            appliesPlan discoverBarterService;
            appliesPlan Negotiating;
            appliesPlan invokeBarterService;}
    invokeBarterService {
            executes barterServiceProcess;
            uses barterServiceGrounding;}
    barterServiceProcess {described_by barterService;}
    barterServiceInterface {presents barterService;}
    barterServiceGrounding {supports barterService;}
}
```

Listing 10. Textual modeling for Agent-SWS Interaction viewpoint of a multi-agent e-barter system.

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

After running Xpand rules for the case study, a JADEX ADF file for barterManager agent and a plan file for each Plan element are generated. Generated ADF file can be used inside a JADEX platform in order to initialize the designed barterManager agent and this agent executes the generated Java plan codes in order to do his tasks.

Part of generated ADF file is shown in Listing 11. In this file, all of the meta-elements and their attributes correspond with related tags. For example Lines 14 to 16 are generated by the template of Listing 9.

```
01  <agent xmlns="http://jadex.sourceforge.net/jadex"
02    …
03    /jadex  http://jadex.sourceforge.net/jadex-2.0.xsd"
04    name= "barterManager" description= "This is
05    barterManager agent" properties= "properties" >
06    <capabilities>
07      <capability>
08        name="barter" file="" description="barter capabilities"
09      </capability>
10    </capabilities>
11    <plans> name="financialPlan"
12        description="financial plan description" priority="1"
13    </plans>
14    <beliefs> name="systemRulesAndEnvironment"
15        description="system rules & environment"
16        dynamic="1"
17    </beliefs>
18    <goals>
19      <achievegoal name="bestMatching" recur=1
20        exclude="when_tried" recalculate="true" retry="true"
21        exported="false" posttoall="false" recurdelay="0"
22        randomselection="false"
23        …
24      </achievegoal>
25    </goals>
26  </agent>
```

Listing 11. Part of generated ADF file from Agent Internal viewpoint of *barterManager* in E-Barter System case study.

Applying Xpand rules, 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 Agent-SWS Interaction viewpoint. ADF and plan files are similar to the one generated for Agent Internal viewpoint. Therefore, only part of the generated OWL-S file is given as an example in Listing 12. Lines from 1 to 6 are boilerplate text that inserted directly from template. barterService, barterServiceInterface, barterServiceProcess and barterServiceGrounding names in lines 8, 17, 26 and 30 in Listing 12 are supplied with «this.name», «int.name», «pro.name» and «gro.name» respectively which are represented in Listing 7.

VII. Conclusion

In this paper, textual concrete syntax of a new DSL, called SEA_L, for Semantic Web enabled MASs is discussed. Additionally, we show 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, automatic application of predefined transformations enables developers to achieve executable code for the agent system that is intended to be implemented in the target platform such as JADEX. Apart from its unique support for Semantic Web, use of SEA_L also brings an easier way of MAS development comparing to merely programming with JADEX or any other specific MAS development framework.

For the concrete syntax, meta-elements are mapped to textual notations, textual constraints are provided and verification of these constraints is illustrated within the instance models. In this way, we provided an interpreter mechanism and made an automatic code generation for users of the domain using Xtext and Xpand tools of Eclipse. For the next step, transformations from SEA_L to other MAS platforms, such as JADE [28] and JACK [19], are aimed.

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf= "&rdf;#" xmlns:rdfs="&rdfs;#"
   xmlns:owl = "&owl;#" xmlns:service= "&service;#"
   …
   xml:base="&DEFAULT;" >
   <owl:Ontology rdf:about="">
      <rdfs:comment> "This ontology represents the OWL-S
         service description for the"+ barterService +
         "service example."
      </rdfs:comment>
      <owl:imports rdf:resource="&service;" />
      …
   </owl:Ontology>
   <service:Service rdf:ID= "barterService">
      <!-- Reference to the Profile -->
      <service:presents rdf:resource="&'barterService_profile;
        # barterServiceInterface"'/>
      …
   </service:Service>
   <!-- Inverse links -->
   <profile:Profile rdf:about=&
        "barterService_profile;# barterServiceInterface">
        <service:presentedBy rdf:resource=#"barterService"/>
   </profile:Profile>
   <process:AtomicProcess rdf:about=&
        "barterService_process;# barterServiceProcess">
         <service:describes rdf:resource=#"barterService"/>
    </process:AtomicProcess>
    <grounding:WsdlGrounding rdf:about=&
        "barterService_grounding;# barterServiceGrounding">
        <service:supportedBy rdf:resource=#"barterService"/>
   </grounding:WsdlGrounding>
</rdf:RDF>
```

Listing 12. Part of generated OWL-S Service file.

REFERENCES

[1] J. M. Bradshaw, “Software Agents,” MIT Press Cambridge, MA, USA, 1997.

[2] K. Sycara, “Multi-agent Systems,” AI Magazine, vol. 19, pp. 79-92, 1998.

[3] N. Shadbolt, W. Hall, T. Berners-Lee, “The Semantic Web Revisited,” IEEE Computer Society, vol. 21(3), pp. 96-101, 2006.

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

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

[6] A. Vallecillo, “A Journey through the Secret Life of Models,” in Perspectives Workshop: Model Engineering of Complex Systems (MECS), 08331 in Dagstuhl Seminar Proceedings, Germany, 2008.

[7] A. Van Deursen, P. Klint, J. Visser, “Domain-specific Languages: an annotated bibliography,” ACM SIGPLAN Notices, vol. 35(6), pp. 26-36, 2000.

[8] M. Mernik, J. Heering, A. Sloane, “When and how to develop domain-specific languages,” ACM Computing Surveys, vol. 37(4), pp. 316-344, 2005.

[9] M. J. Varanda-Pereira, M. Mernik, D. Da Cruz, P. R. Henriques “Program comprehension for domain-specific languages,” Computer Science Information Systems, vol. 5(2), pp. 1-17, 2008.

[10] M. Fowler, “Domain-specific Languages,” Addison Wesley, 2011.

[11] S-H. Liu, A. Cardenas, M. Mernik, B. R. Bryant, J. Gray, X. Xiong, “Introducing domain-specific language implementation using web-service oriented technologies”, Multiagent and Grid Systems-An International Journal, vol. 8, pp. 19-44, 2012.

[12] M. Challenger, S. Getir, S. Demirkol, G. Kardas, “A Domain Specific Metamodel for Semantic Web enabled Multi-agent Systems,” Lecture Notes in Business Information Processing, vol. 83, pp. 177-186, 2011.

[13] U. Kulesza, A. Garcia, C. Lucena and P. Alencar, “A Generative Approach for Multi-agent System Development,” Lecture Notes in Computer Science, vol. 3390, pp. 52-69, 2005.

[14] S. Rougemaille, F. Migeon, C. Maurel, M-P. Gleizes, “Model Driven Engineering for Designing Adaptive Multi-agent Systems,” Lecture Notes in Artificial Intelligence, vol. 4995, pp. 318-33, 2007.

[15] C. Hahn, “A Domain Specific Language for Multi-agent Systems,” in 7th Int. Conf. on Autonomous Agents and Multi-agent Systems (AAMAS’08), ACM Press, pp. 233-240, 2008.

[16] C. Hahn, S. Nesbigall, S. Warwas, I. Zinnikus, K. Fischer, M. Klusch, “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, pp. 200-206, 2008.

[17] M. E. Bratman, “Intention, Plans, and Practical Reason,” Harvard University Press: Cambridge, Massachusetts, 1987.

[18] JADEX: http://jadex-agents.informatik.uni-hamburg.de/xwiki/bin/view/ About/Overview

[19] JACK: http://aosgrp.com/products/jack/index.

[20] OWL-S: http://www.w3.org/Submission/OWL-S/ OMG ODM: http://www.omg.org/spec/ODM/1.0/

[21] OWL: http://www.w3.org/TR/owl-features

[22] Xtext: http://www.eclipse.org/Xtext/

[23] ISO/IEC 14977:1996 Standard, “Information technology, Syntactic meta language - Extended BNF,”

[24] Xpand: http://wiki.eclipse.org/Xpand

[25] Eclipse EMF: http://www.eclipse.org/modeling/emf

[26] Xpand documentation: http://ditec.um.es/ssdd/xpand_reference.pdf

[27] JADE: Java Agent DEvelopment Framework, http://jade.tilab.com/

[28] S. Demirkol, S. Getir, M. Challenger and G. Kardas, “Development of an Agent based E-barter System,” International Symposium on Innovations in Intelligent Systems and Applications (INISTA), IEEE Computer Society, pp. 193-198, 2011.